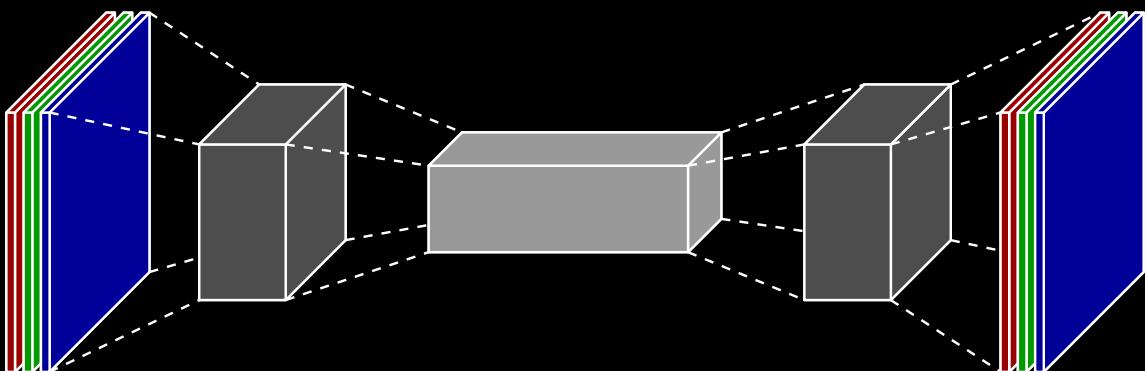


MATURAARBEIT

Maschinelles Lernen mit TensorFlow

Entwicklung eines Convolutional Denoising Autoencoders



Luis Wirth

GYMNASIUM OBERWIL
Klasse 4a

Betreut von
Dr. Jonas GLOOR

Abgegeben im September 2019

Inhaltsverzeichnis

Inhaltsverzeichnis	I
Vorwort	III
Einleitung	IV
Konvention und Notation	V
1 Maschinelles Lernen	1
1.1 Allgemeine Begriffe	1
1.1.1 Daten	1
1.1.2 Modelle	2
1.2 Training	3
1.2.1 Verlust- und Kostenfunktionen	3
1.2.2 Stochastisches Gradientenverfahren	4
1.3 Trainingsphänomene	5
1.3.1 Konvergenz und Divergenz	5
1.3.2 Underfitting und Overfitting	5
2 Deep Learning und Künstliche Neuronale Netze	7
2.1 Perzeptron	7
2.1.1 Lernpotenzial eines Perzeptrons	8
2.2 Erweiterung der künstlichen Neuronen	8
2.2.1 Künstliche Neuronen im Allgemeinen	8
2.2.2 Perzeptronen als künstliche Neuronen	9
2.2.3 ReLU-Neuronen	9
2.2.4 Sigmoid-Neuronen	10
2.3 Topologie der Künstlichen Neuronalen Netze	10
2.4 Lernverhalten	11
2.5 Vorwärtspropagierung	11
2.5.1 Initialisierung der Modellparameter	14
2.6 Rückwärtspropagierung	15
2.7 Universal Approximation Theorem	15
3 Convolutional Neural Networks	16
3.1 Bilder als Tensoren	16
3.2 Topologie der Convolutional Neural Networks	17
3.3 Convolutional-Schichten und Filter	18
3.3.1 Filter in der Bildverarbeitung	18
3.3.2 Filter in CNNs	19
3.3.3 Filteroperation intuitiv	19
3.3.4 Filteroperationen als diskrete Faltungen	20
3.3.5 Mehrere Filter	22
3.3.6 Padding	22
3.3.7 Stride	23
3.3.8 Vorzüge von Filtern	24
3.3.9 Convolutional-Schicht	24
3.4 Dimensionalitätskontrolle	25
3.4.1 Pooling-Schicht	25
3.4.2 Upsampling-Schicht	26

4 Autoencoder	27
4.1 Topologie	27
4.2 Funktionsweise	28
4.3 Anwendungen	29
4.3.1 Datenkompression	29
4.4 Convolutional-Autoencoder	30
4.5 Denoising-Autoencoder	30
4.5.1 Generierung der verrauschten Bilder	31
5 Frameworks für Maschinelles Lernen	32
5.1 TensorFlow	32
5.1.1 Frontend und Backend	32
5.1.2 Graph	33
5.1.3 Ausführung	34
5.1.4 Tensorboard	35
5.2 Keras	35
5.2.1 Sequential-Modell	36
5.2.2 Schichten	36
5.2.3 Training und Evaluierung	37
6 Entwicklung eines Denoising-Autoencoders	39
6.1 Das konkrete Modell	39
6.1.1 Daten	39
6.2 Setup	40
6.3 Entwicklung	41
6.3.1 Testprogramm	41
6.3.2 Trainingsdaten	42
6.3.3 Modell definieren	43
6.3.4 Modell trainieren	45
6.3.5 Modell ausführen	46
6.3.6 Hyperparameter einstellen	47
6.3.7 Diskussion des Modells	48
Rückblick und Ausblick	51
Schlusswort	54
A Gradientenverfahren	55
B Herleitung der Rückwärtspropagierung für KNNs	58
C Performance von TensorFlow	63
D GitHub	65
Literatur	66
Abbildungsverzeichniss	67
Tabellenverzeichniss	69
Selbstständigkeitserklärung	70

Vorwort

Persönliche Themenwahl

Maschinelles Lernen (machine learning, kurz: ML) stösst seit einigen Jahren auf grosses Interesse sowohl in der Wissenschaft, als auch in der Wirtschaft. Dies motivierte mich, Näheres im Selbststudium darüber in Erfahrung bringen zu wollen. Bereits im Jahr 2017 begann ich, ein erstes eigenes Programm zu entwickeln, welches einen Genetischen Algorithmus implementierte (Evolutionäre Entwicklung von lernfähigen Agenten). Diese erste Erfahrung öffnete mir die Tür für ein Praktikum am Departement für Computational Sciences der Universität Basel. Dort durfte ich ein Künstliches Neuronales Netz in C++ programmieren, welches ich dann auf eine konkrete physikalische Problemstellung anwenden konnte (Vorhersage zu den Energiezuständen von Wassermolekülen in verschiedenen atomaren Konfigurationen). Dieses erste Praktikum hatte mich dazu motiviert, mich tiefgehender mit der Thematik zu befassen und die grundlegende Funktionsweise von ML durchdringen zu wollen.

Daraufhin entstand die Idee, die vorliegende Arbeit zu verfassen. Auf diese Weise wollte ich mir ein ausgeprägteres mathematisches Verständnis für Maschinelles Lernen aneignen und auch fortgeschrittene Modelle betrachten. Zudem hatte ich ebenfalls schon von den verbreiteten Deep-Learning-Frameworks TensorFlow und Keras gehört, welche ich unbedingt erlernen wollte. Daher entschied ich mich, einen Convolutional-Autoencoder in TensorFlow zu programmieren und vorher die nötige Theorie zu erklären. Zuerst verfolgte ich das Ziel, diesen Autoencoder als ein Generatives Modell zu verwenden, um Bilder von künstlich generierten menschlichen Gesichter zu erzeugen.

Während meiner Sommerferien konnte ich ein 6-wöchiges Praktikum im Forschungszentrum von Adobe in San Francisco leisten, wo ich ebenfalls ein ML-Modell programmiert habe (Nutzung von Gesichtserkennung für Marketing-Zwecke). Durch meine praktikumsbezogenen Recherchen gelangte ich jedoch schnell zu der Erkenntnis, dass mein ursprüngliches ambitioniertes Vorhaben bezüglich der generativen Nutzung eines Autoencoder den vorgegebenen Rahmen einer Maturaarbeit deutlich sprengen würde. Ich hätte nämlich nicht nur einen Autoencoder programmieren müssen, sondern sogar einen Variational Autoencoder, welcher mathematisch noch anspruchsvoller gewesen wäre.

Aus diesem Grund beschloss ich, meine Leitfrage in Abstimmung mit meiner Betreuungsperson anzupassen. Diese beschränkt sich nunmehr auf die eigene Entwicklung eines Convolutional *Denoising* Autoencoder in TensorFlow und Keras sowie vorgängig die zugrunde liegende Theorie zu erklären.

Danksagung

Mein herzlicher Dank gilt dem Department für Computational Sciences der Universität Basel, namentlich Herrn Professor Stefan Goedecker, weil er und sein Team mir die ersten wissenschaftlichen Einblicke in das Thema Machine Learning eröffnet haben.

Danken möchte ich ebenso der Firma Adobe, welche mir ermöglicht hat, im Rahmen eines 6-wöchigen Praktikums die kommerzielle Anwendung von Machine Learning vertieft kennenzulernen.

Auch danke ich meinen Eltern, Frau Dr. Doris Fellenstein und Herrn Dr. Victor Wirth, für ihre Begleitung und Unterstützung während des Erstellungsprozesses meiner Maturaarbeit.

Zu guter Letzt gilt mein besonderer Dank meiner Betreuungsperson, Herrn Dr. Jonas Gloor, für seine wertvollen fachlichen Inputs.

Einleitung

Maschinelles Lernen erfreut sich aktuell grosser Beliebtheit und wird sehr erfolgreich in naturwissenschaftlichen, medizinischen oder auch wirtschaftlichen Themenbezügen angewendet. Es bietet ein enormes Innovationspotenzial und wird laufend weiterentwickelt. Obwohl das Schlagwort "Künstliche Intelligenz" in aller Munde ist, wissen die wenigsten, wie sie tatsächlich funktioniert.

Das Ziel dieser Arbeit ist daher, dem Leser ein umfassendes Grundverständnis über Maschinelles Lernen zu vermitteln, sowie auf dieser Basis ein konkretes Anwendungsbeispiel zu entwickeln. Zu diesem Zweck werden zunächst die theoretischen Grundlagen zum modellbasierten Lernen dargelegt. Darauf aufbauend ist es möglich, ein konkretes Modell, namentlich Künstliche Neuronale Netze (KNN), in ihrer Funktionsweise zu erläutern. In logischer Konsequenz wird anschliessend eine spezifische Architektur eines KNNs präsentiert. Namentlich handelt sich um ein Convolutional Neural Networks (CNN), welches sich besonders gut für Bildverarbeitungszwecke eignet und daher eine grosse Verbreitung in der Praxis aufweist. Mit der anschliessenden Erläuterung der mathematischen Beschreibung eines Autoencoders sind die theoretischen Grundlagen für ein konkretes Anwendungsbeispiel geschaffen.

Nachdem ein mathematisch fundiertes Verständnis für Maschinelles Lernen vermittelt wurde, sollen im Weiteren zwei Open-Source Implementationen vorgestellt werden. Dabei handelt es sich um TensorFlow, ein ursprünglich von Google entwickeltes Framework für Deep-Learning, und um Keras, eine übergeordnete Schnittstelle für eine vereinfachte TensorFlow-Verwendung. Neben ihrer grossen Verbreitung besitzen sie den Vorteil eines verkürzten Einstiegs in die ML-Programmierung.

Für das konkrete Modell werden wir einen sogenannten Convolutional-Denoising-Autoencoder entwickeln. Mit dessen Hilfe kann das Bildrauschen von Bildern entfernt werden.

Die Leitfrage lässt sich demnach folgendermassen formulieren:

Wie funktioniert Maschinelles Lernen, aufgezeigt anhand eines konkreten Modells, unter Verwendung von TensorFlow und Keras?

Dabei lassen sich folgende Teilschritte unterscheiden:

1. Erarbeiten der allgemeinen Theorie zum Maschinellen Lernen
2. Künstliche Neuronale Netze als Modelle für ML
3. Convolutional Neural Networks und Autoencoder als konkrete KNN-Architekturen
4. Grundsätzliche Funktionsweise der Frameworks TensorFlow und Keras
5. Programmierung eines konkreten Modells mit TensorFlow und Keras

Konvention und Notation

Beschriftung

Zahlen und Tensoren

a	ein Skalar (Zahl)
\mathbf{a}	ein Vektor
\mathbf{A}	eine Matrix
\mathbf{A}	ein Tensor

Mengen

(a, b)	ein geordnetes Paar (2-Tupel) der Elemente a und b
\mathbb{A}	eine Menge
\mathbb{R}	die Menge aller reellen Zahlen
$\{a, b\}$	die Menge, welche aus den Elementen a und b besteht
$\{1, \dots, n\}$	die Menge aller ganzen Zahlen von 1 bis n
$a \in \mathbb{A}$	a ist ein Element der Menge \mathbb{A}
\mathbb{R}^n	ein n -dimensionale Vektorraum

Indexierung

a_i	die i -te Vektorkomponente (Indexierung beginnt mit 1)
$(\mathbf{A})_{i,j}$	das Matrixelement in der i -ten Zeile und der j -ten Spalte
$A_{i,j}$	das Matrixelement in der i -ten Zeile und der j -ten Spalte
$\mathbf{A}_{i,:}$	die Zeile i der Matrix \mathbf{A}
$\mathbf{A}_{:,i}$	die Spalte i der Matrix \mathbf{A}
$\mathbf{A}_{i,:,:}$	der i -te Querschnitt entlang der Höhe des Tensors \mathbf{A}
$\mathbf{A}_{:,:,i}$	der i -te Querschnitt entlang der Breite des Tensors \mathbf{A}
$\mathbf{A}_{:::,i}$	der i -te Querschnitt entlang der Tiefe des Tensors \mathbf{A}

Lineare Algebra Operationen

$\Sigma(\mathbf{A})$	die Summe aller Elemente von Matrix \mathbf{A}
$v \cdot w$	das Skalarprodukt von v mit w
\mathbf{A}^\top	das Transponierte einer Matrix \mathbf{A}
$\mathbf{A} \odot \mathbf{B}$	das elementweise (Hadamard) Produkt
$\mathbf{A} * \mathbf{B}$	die diskrete Faltung von \mathbf{A} ueber \mathbf{B}

Infinitesimalrechnung

$f'(x)$	die Ableitung der Funktion f bezüglich seinem Argument x
$\frac{dy}{dx}$	die Ableitung von y bezüglich x
$\frac{\partial y}{\partial x}$	die partielle Ableitung von y bezüglich x
∇y	der Gradient von y
$\nabla_x y$	der Gradient von y bezüglich x (Vektor)
$\int f(x) dx$	das unbestimmte Integral der Funktion f bezüglich x
$\int_a^b f(x) dx$	das bestimmte Integral der Funktion f bezüglich x von a bis b
$\lim_{x \rightarrow a} f(x)$	der Limes/Grenzwert von f , wenn x gegen a strebt

Funktionen

$f : \mathbb{A} \rightarrow \mathbb{B}$	eine Funktion f mit Definitionsmenge \mathbb{A} und Zielmenge \mathbb{B}
$f(x)$	eine Funktion f mit Argument x (Skalar)
$f(\mathbf{v})$	eine Funktion f mit Argument \mathbf{v} (Vektor)
$\mathbf{f}[\mathbf{v}]$	die vektorisierte Funktion \mathbf{f} mit Argument \mathbf{v} (Vektor)
\mathcal{A}	ein Funktionenraum
$f * g$	die Faltung von f über g
$f \circ g$	die Komposition von den Funktionen f und g

Statistik

\mathcal{N}	die Gauss'sche Normalverteilung
ϕ	die Gauss'sche Dichtefunktion
μ	der Erwartungswert/Mittelwert
σ	die Standardabweichung
σ^2	die Varianz

Sonstiges

$\sum_{i=a}^b x$	die Summe mit von a bis b ueber x mit Laufvariable i
------------------	--

Kapitel 1

Maschinelles Lernen

Im ersten Kapitel sollen die Grundlagen für ein fundiertes Verständnis von Maschinellem Lernen gelegt werden. Neben den verschiedenen Arten sollen essenzielle Begriffe und das Modellkonzept dargelegt werden. Darüber hinaus werden die Funktionsweise des Trainings sowie daraus entstehende Phänomene erläutert. Auf dieser Basis erfolgt in Kapitel zwei die Darlegung eines konkreten Modells.

Maschinellen Lernens beschäftigt sich mit Algorithmen und mathematischen Modellen, welche die Fähigkeit entwickeln, Probleme selbstständig zu lösen. Hierbei wird nicht explizit einprogrammiert, wie ein Modell das Problem zu lösen hat, stattdessen wird das Modell trainiert, optimiert sich von selbst und findet allein einen Weg, die Aufgabenstellung zu lösen. Die Grundidee dabei ist, dass Daten erfasst, generiert oder gemessen werden, welche anschliessend analysiert werden sollen. Innerhalb dieser Daten existieren gewisse Gesetzmässigkeiten und Muster. Diese Muster sollen vom Modell erkannt und verallgemeinert werden. Nach dem erfolgreichen Lernprozess kann das Modell Vorhersagen für neue Daten machen.

Wir unterscheiden zwei Arten von Maschinellem Lernen¹:

- **Überwachtes Lernen** (engl.: supervised learning) ist ein Lernverfahren, bei welchem die Daten aus zwei Teilen bestehen, aus Inputs und Outputs. Man bezeichnet dabei die Outputs als Labels. Die Aufgabe des Modells ist es, eine **Korrelation** zwischen den Inputs und den Labels zu erlernen und so ihre Beziehung zueinander zu verstehen. Anhand der Informationen, welche die Inputs enthalten, können die Labels vorhergesagt werden. Die vorhergesagten Labels des Modells werden mit den wahren Labels abgeglichen. Mit dieser Überwachung werden die Fähigkeiten des Modells bewertet.

Voraussetzung dafür ist, dass die Daten “gelabelt” sein müssen. Daher müssen bereits vorgängig Daten vorhanden sein, welchen die gewünschten Labels aufweisen. Zudem muss auch die erwähnte Korrelation bestehen. Falls kein Zusammenhang zwischen den Inputs und den Outputs existiert, kann das Modell auch keine Vorhersagen machen und damit auch nichts erlernen.

- **Unüberwachtes Lernen** (engl.: unsupervised learning) ist ein anderes Lernverfahren, bei welchem diese Labels nicht vorhanden sind. Dem Modell stehen nur die Inputdaten zur Verfügung. Diese werden ebenfalls analysiert und das Modell soll Muster extrahieren, welche sich von einem zufälligen Datenrauschen unterscheiden.

Der Grossteil der Modelle des Maschinellen Lernens zählt zum Überwachten Lernen, da es mehr Anwendungsmöglichkeiten bietet. Grundsätzlich steht daher das Überwachte Lernen im Vordergrund der Arbeit. Allerdings gehört der später erläuterte Autoencoder (siehe Sektion (4)) zum Unüberwachten Lernen. Jedoch folgt dieser — im Gegensatz zu sonstigen Unüberwachten Modellen — der gleichen Systematik wie das Überwachten Lernen.

Quellen: (1) (2)

1.1 Allgemeine Begriffe

1.1.1 Daten

Um ein Modell zu trainieren, werden Daten benötigt. Diese Daten bestehen immer aus **Inputs** und **Outputs**. Man unterscheidet hierbei zwischen zwei Arten von Outputs. Die **Labels** sind die erwarteten Outputs, welche

¹Es existieren noch weitere Arten von ML, wie zum Beispiel **Semi-Supervised Learning** und **Reinforcement Learning**. Für diese Arbeit sind diese aber nicht weiter relevant.

die gewünschten Zielwerte sind. Die **Vorhersagen** sind die Outputs, die vom Modell produziert werden und hoffentlich möglichst genau mit den Labels übereinstimmen.

Diese Daten für das Training kommen in der Form eines **Trainingsdatensatzes** \mathbb{X} . Es handelt sich um eine Menge an **Samples**, welche jeweils aus Inputs und Labels bestehen. Die Inputs werden in einem Vektor

$$\mathbf{x} = (x_1 \quad x_2 \quad \cdots \quad x_m)^\top$$

und die Labels in einem Vektor

$$\hat{\mathbf{y}} = (\hat{y}_1 \quad \hat{y}_2 \quad \cdots \quad \hat{y}_n)^\top$$

zusammengefasst. Somit ist ein Trainingssample ein Paar $(\mathbf{x}_i, \hat{\mathbf{y}}_i)$ bestehend aus einem Inputvektor \mathbf{x} und einem Labelvektor $\hat{\mathbf{y}}$. Die Vorhersagen werden ebenfalls in einem Vektor

$$\mathbf{y} = (y_1 \quad y_2 \quad \cdots \quad y_n)^\top$$

zusammengefasst.

Die Inputs beinhalten sogenannte **Features** (deutsch: Merkmale). Sie zeichnen die Inputs aus und umfassen ihren gesamten Informationsgehalt. Der Algorithmus soll anhand dieser Features seine Vorhersagen machen. Diese Vorhersagen werden dann mit den Labels abgeglichen und bewertet. Anhand der Bewertungen wird eine Optimierung des Modells vorgenommen. Unter korrekten Bedingungen (kein Overfitting (siehe (1.3.2))) findet kein Auswendiglernen der Trainingsdaten statt, sondern ein Generalisieren des Zusammenhangs anhand von Mustern und Gesetzmäßigkeiten.

Um eine endgültige Bewertung des Modells durchzuführen, wird ein Testdatensatz \mathbb{T} genutzt, um Vorhersagen zu generieren. Dieser ist nicht Teil des Trainingdatensatzes. Er garantiert also, dass kein Auswendiglernen möglich ist.

Beispiel für Modelldaten

Um die Begriffe besser verständlich zu machen, folgt nun ein Beispiel: Ein Modell wird darauf trainiert, die Grösse einer Person anhand von Alter und Gewicht abzuschätzen. Der Trainingsdatensatz besteht aus mehreren Samples, wobei jedes Sample Werte enthält, welche die Messdaten einer Person verkörpern. Ein Sample besteht beispielsweise aus dem Input $\mathbf{x} = (18 \text{ yr} \quad 65 \text{ kg})^\top$ und dem Label $\hat{\mathbf{y}} = (180 \text{ cm})^\top$. Das trainierte Modell erzeugt für den Input dieses Samples die Vorhersage $\mathbf{y} = (176 \text{ cm})^\top$.

Quellen: (3) (2)

1.1.2 Modelle

Ein **Modell** ist eine mathematische Funktion $h: \mathbb{R}^m \rightarrow \mathbb{R}^n$, Hypothesenfunktion genannt, welche die Inputs auf die Outputs abbildet $\mathbf{y} = h(\mathbf{x})$. Man kann diese Funktion als die Hypothese auffassen, welche das Modell bezüglich der Beziehung zwischen den Inputs und den Labels aufgestellt hat. Ein solches Modell kann verwendet werden, um entweder Klassifizierungsprobleme oder Regressionsprobleme zu lösen. Falls es sich um Letzteres handelt, spricht man von einem Regressionsmodell. Ein solches Regressionsproblem soll im Rahmen dieser Arbeit gelöst werden.

Das Verhalten eines Modells bestimmt sich durch seine **Modellparameter** $\lambda_1, \lambda_2, \dots, \lambda_k$. Sie determinieren, wie die Hypothese des Modells lautet. Das Ziel ist es, die Modellparameter so einzustellen, dass die Vorhersagen \mathbf{y} des Modells möglichst exakt mit den Labels $\hat{\mathbf{y}}$ der Trainingsdaten übereinstimmen. Dies wird iterativ gemacht, indem immer wieder leichte Anpassungen an den Parametern vorgenommen werden, bis das Modell die gewünschten Resultate liefert.

Neben den gelernten Parametern gibt es auch noch sogenannte **Hyperparameter**. Diese können nicht erlernt werden, sondern müssen manuell vor dem Training gewählt werden und können den Lernvorgang erheblich beeinflussen. Dies bedeutet, dass man ausprobieren muss, welche Werte der Hyperparameter die besten Resultate liefern.

Für Maschinelles Lernen haben sich gewisse Modelle besonders gut etabliert, dazu zählen: Support Vector Machines, Evolutionäre Algorithmen, und Künstliche Neuronale Netze. Diese Arbeit wird sich vorwiegend mit Neuronalen Netzen auseinandersetzen.

Beispiel für ein Modell

Es wird nun erneut ein Beispiel erläutert, um das Konzept zu verdeutlichen. Es soll ein Modell entwickelt werden, um den Zusammenhang zwischen dem Benzinverbrauch eines Autos und der zurückgelegten Strecke zu erlernen. Dafür wird angenommen, dass es sich dabei um eine lineare Beziehung zwischen diesen Variablen handelt. Somit eignet sich das wohl einfachste Regressionsmodell: eine Regressionsgerade der Form $y = \lambda_1 x + \lambda_0$, wobei y der Benzinverbrauch in Litern und x die Strecke in Kilometern ist. Beim Trainieren werden die besten Werte für die Parameter λ_0 und λ_1 gesucht, welche die Vorhersagen am besten mit den Labels übereinstimmen lässt. Das Modell könnte somit nach dem Training folgende Hypothesenfunktion besitzen $y = h(x) = 0.07 \cdot x + 0$. Dies entspricht der Hypothese, dass ein Auto einen Spritverbrauch von 7 Litern pro 100 Kilometer hat.

Quellen: (2)

1.2 Training

1.2.1 Verlust- und Kostenfunktionen

Einsicht ist der erste Schritt zur Besserung. Diese Maxime gilt auch für das Machine Learning. Deshalb muss beim Training zuerst die Genauigkeit des Modells bewertet werden. Dies wird mithilfe von sogenannten Kostenfunktionen bzw. Verlustfunktionen erreicht.

Eine **Verlustfunktion** $L(y, \hat{y})$ soll ein Mass für die Abweichung der Vorhersage y von dem Label \hat{y} sein. Aus ihr bildet man die **Kostenfunktion** $C(\mathbf{y}, \hat{\mathbf{y}})$, indem die Verluste der einzelnen Outputs aufsummiert werden. Somit erhält man die Kosten einer gesamten Vorhersage mit m Outputs (siehe Gl. (1.1)). Bei gewissen Kostenfunktionen gibt es keine Verlustfunktion, da sich diese nicht auf die einzelnen Outputpaare aufgeteilt werden kann. Der Fehler $\bar{C}(\mathbb{X})$ des gesamten Trainingsdatensatzes \mathbb{X} , der Grösse p , ergibt sich aus dem arithmetischen Mittel der einzelnen Kosten der Vorhersagen (siehe Gl. (1.2)).

$$C(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^m L(y_i, \hat{y}_i) \quad (1.1) \qquad \bar{C}(\mathbb{X}) = \frac{1}{p} \sum_{j=1}^p C(\mathbf{y}_j, \hat{\mathbf{y}}_j) \quad (1.2)$$

Eine Kostenfunktion C sollte folgende Eigenschaften aufweisen:

- C ist minimal, wenn $\mathbf{y} = \hat{\mathbf{y}}$
- C wächst mit $|\mathbf{y} - \hat{\mathbf{y}}|$
- C ist nach jedem y_n partiell differenzierbar (erklärt in Anhang (A))

Quellen: (3) (2)

1.2.1.1 Mittlere quadratische Abweichung

Die bekannteste Kostenfunktion ist die "Mittlere quadratische Abweichung" (engl.: mean squared error, kurz: MSE). Sie ist definiert als das arithmetische Mittel aller quadrierten Differenzen zwischen den Vorhersagen und den Labels. Zusätzlich halbiert man noch das arithmetische Mittel, damit bei der Ableitung der Faktor 2 wegfällt. Sie hat keine entsprechende Verlustfunktion, da auf diese Weise das arithmetische Mittel nicht ausgedrückt werden kann. Die Kostenfunktion kann mithilfe einer Summe berechnen werden oder mithilfe einer Vektorsubtraktion der Vorhersagen \mathbf{y} minus den Labels $\hat{\mathbf{y}}$.

$$C_{\text{MSE}} = \frac{1}{2n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 = \frac{1}{2n} (\hat{\mathbf{y}} - \mathbf{y})^2 \quad (1.3)$$

Sie erfüllt alle Anforderungen an eine Kostenfunktion:

- Ihr Funktionswert ist 0 und minimal für $\mathbf{y} = \hat{\mathbf{y}}$
- Sie ist proportional zu $(\hat{\mathbf{y}} - \mathbf{y})^2$
- Ihre partielle Ableitung nach einem y_i lautet: $C'_{\text{MSE}} = \frac{1}{n}(y_i - \hat{y}_i)$

Quellen: (3)

1.2.2 Stochastisches Gradientenverfahren

Beim Training eines Modells handelt es sich um ein Optimierungsproblem. Das Modell macht die besten Voraussagen, wenn die Funktionswerte der Kostenfunktion am kleinsten sind. Deshalb muss die gewählte Kostenfunktion C minimiert werden. Hierbei muss die Funktion C nicht mehr in Abhängigkeit der Inputs und Outputs betrachtet werden, sondern als Funktion der Modellparameter $C(\lambda_1, \lambda_2, \dots, \lambda_k)$. Denn diese Werte sollten angepasst werden, um das Modell zu verbessern.

Für diese Optimierung wird das sogenannte **Gradientenverfahren** (engl.: Gradient descent) verwendet. Das Gradientenverfahren ist eine gängige Methode, um Funktionen $f : \mathbb{R}^n \rightarrow \mathbb{R}$ zu minimieren². Falls dem Leser das Prinzip nicht vertraut ist, wird auf Anhang (A) verwiesen, in welchem die Funktionsweise des Gradientenverfahrens erklärt wird.

Das Gradientenverfahren zur Minimierung verläuft iterativ nach folgender Gleichung:

$$\lambda_{t+1} = \lambda_t - \eta \cdot \nabla C(\lambda_t) \quad (1.4)$$

Wie die initialen Modellparameter $\lambda_{t=0}$ zu wählen sind, wird in Sektion (2.5.1) erläutert.

Die sogenannte **Lernrate** η aus Gleichung (1.4) stellt einen Hyperparameter dar. Sie ist der positive Proportionalitätsfaktor, welcher die Schrittgrösse des Gradientenabstiegs bestimmt. Je nach zu minimierender Funktion muss sie anders gewählt werden. Dies geschieht durch Ausprobieren. Falls η nicht gut gewählt wurde, ergeben sich Probleme beim Training:

- Falls η zu klein ist, verläuft das Training unnötig langsam. Außerdem kann es passieren, dass die Optimierung bei einem hohen lokalen Minimum stecken bleibt.
- Falls η zu gross ist, kann es passieren, dass man über das lokale Minimum hinaus schiesst und somit nur darum herum springt (siehe Divergenz (1.3.1)).

Aus nachfolgend erläuterten Gründen wird für ML ein etwas angepasstes Verfahren des Gradientenabstiegs verwendet: das **Stochastische Gradientenverfahren** (SGD). Das Problem des herkömmlichen Gradientenverfahrens besteht darin, dass der Gradient für den *gesamten* Trainingsdatensatz berechnet werden muss. Dies ist zwar ein exakter Prozess, aber ein extrem langsamer zugleich. Bei grossen Datensätzen würde es sehr lange dauern, bis das Modell nur annähernd gute Voraussagen machen könnte. Somit steht die Genauigkeit in keinem Verhältnis zur Effizienz dieser Methode.

Bei SGD wird der “echte” Gradient des gesamten Datensatzes mit dem Gradient einiger Trainingssamples approximiert. Dazu wird der Trainingsdatensatz in sogenannte **Mini-Batches** eingeteilt und der Gradient jeweils pro Mini-Batch berechnet. Als Konsequenz finden deutlich mehr Iterationen in einer einzigen Durchkämmung der Trainingsdaten statt. Eine solche vollständige Durchkämmung der Trainingsdaten bezeichnet man als eine **Epoche**. Oft wird mehrere Epochen lang trainiert, bis das Modell genügend gute Resultate liefert. Jedoch sollte auch nicht zu oft mit den gleichen Daten trainiert werden, da es sonst zu Overfitting (siehe Sektion (1.3.2)) kommen kann. Der Gradient eines genug grossen Mini-Batches ist zwar nicht ganz exakt, aber approximiert den Gradienten des gesamten Datensatzes genügend gut. Sowohl die Mini-Batch Grösse, wie auch die Anzahl Epochen sind weitere Hyperparameter.

Die partiellen Ableitungen der gesamten Trainingsdaten werden mit dem arithmetischen Mittel der partiellen Ableitungen eines Mini-Batches der Grösse q approximiert.

$$\frac{\partial \bar{C}}{\partial \lambda_k} \approx \frac{1}{q} \sum_{i=1}^q \frac{\partial C_i}{\partial \lambda_k} \quad (1.5)$$

Eine Iteration des Stochastischen Gradientenverfahrens wird analog zu Gleichung (1.4) folgendermassen durchgeführt.

$$\lambda_{k,t+1} = \lambda_{k,t} - \frac{\eta}{q} \sum_{i=1}^q \frac{\partial C_i}{\partial \lambda_{k,t}} \quad (1.6)$$

Quellen: (3) (2)

²Üblicherweise wird auf Gymnasialstufe vermittelt, dass die lokalen Extrema einer Funktion f bestimmt werden können, indem die erste Ableitung f' gebildet und gleich null gesetzt wird. Dies ist hier jedoch nicht möglich, da die Funktion $C'(\lambda_1, \lambda_2, \dots, \lambda_n)$ zu kompliziert ist, um die Nullstellen analytisch zu bestimmen. Deshalb wird hier das numerische Gradientenverfahren verwendet.

1.3 Trainingsphänomene

1.3.1 Konvergenz und Divergenz

Beim Training eines Modells kann es entweder zu **Konvergenz** oder **Divergenz** kommen. Falls die Vorhersagen im Verlaufe des Trainings immer besser mit den Labels übereinstimmen, bzw. die Kostenfunktion immer kleiner wird, gilt das Modell als konvergierend. Also findet das Gradientenverfahren erfolgreich ein lokales Minimum.

Jedoch kommt es auch vor, dass ein Modell nicht konvergiert oder vielleicht sogar divergiert. Bei Divergenz werden die Kosten im Verlaufe des Training immer grösser. Dies kann verschiedene Gründe haben. Einige davon können sein:

- zu grosse Lernrate η
- sonstige falsche Hyperparameter
- falsches Modell
- zu wenig Trainingsdaten
- zu schwache Korrelation zwischen Inputs und Labels

1.3.2 Underfitting und Overfitting

Falls ein Modell konvergiert, heisst das noch nicht, dass es die Gesetzmässigkeiten innerhalb der Trainingsdaten richtig erlernt hat. Im Wesentlichen kann es zu zwei Problemen kommen: Overfitting oder Underfitting.

Overfitting bezeichnet das Phänomen, dass ein Modell zwar Vorhersagen erzeugt, welche jedoch zu stark an die gegebenen Trainingssamples angepasst sind. Dies liegt zumeist daran, dass das Modell zu viele Parameter besitzt oder aber der Trainingsdatensatz zu wenige Samples dafür beinhaltet. Somit übersteigt die Komplexität des Modells gewissermassen jene der Aufgabenstellung.

Um das Phänomen zu verdeutlichen, wird ein ganzrationales Regressionsmodell betrachtet. Dieses besitzt eine Hypothesenfunktion $h = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$, welche ein Polynom n -ter Ordnung ist. Falls der Trainingsdatensatz nun aus n oder weniger Samples besteht, kann das Modell die Regressionskurve der Hypothesenfunktion exakt durch jeden Datenpunkt legen. Dies entspricht dem Verhalten eines Modells mit n Modellparametern, welches mit n Samples trainiert wird, und dabei exakt jedes Samples *auswendig* lernt, anstatt dessen Gesetzmässigkeiten zu erkennen.

Beim Auswendiglernen misst das Modell dem Datenrauschen zu viel Bedeutung zu, welches durch die natürliche Varianz innerhalb der Trainingsdaten entsteht. Somit nutzt es irrelevante Modellparameter, um das Rauschen zu kopieren. Dadurch kann das Modell zwar sehr gute Vorhersagen zum Trainingsdatensatz \mathbb{X} machen, jedoch würde es schlechte Vorhersagen für einen anderen Testdatensatz \mathbb{T} liefern, welchen es nicht auswendig lernen konnte.

Somit kann Overfitting folgendermassen definiert werden: Eine Hypothesenfunktion h overfittet dann, wenn eine alternative Hypothesenfunktion h' existiert, für welche die Kosten bezüglich dem Trainingsdatensatz grösser sind $C_{h'}(\mathbb{X}) > C_h(\mathbb{X})$, jedoch die Kosten für Testdatensatz kleiner sind $C_{h'}(\mathbb{T}) < C_h(\mathbb{T})$.

Das Gegenteil von Overfitting ist **Underfitting**. Dabei handelt es sich um das Phänomen, dass eine Hypothesenfunktion h zu wenige Modellparameter λ_k besitzt, um die Komplexität der Aufgabenstellung zu bewältigen. Die Parameter reichen nicht aus, damit das Modell die Korrelation zwischen Inputs und Labels begreifen kann.

Bezogen auf das ganzrationale Regressionsmodell bedeutet dies, dass der Grad n der polynomen Hypothesenfunktion h zu gering ist, um sich an die Datenpunkte der Samples anzuschmiegen.

Quellen: (4) (1) (2)

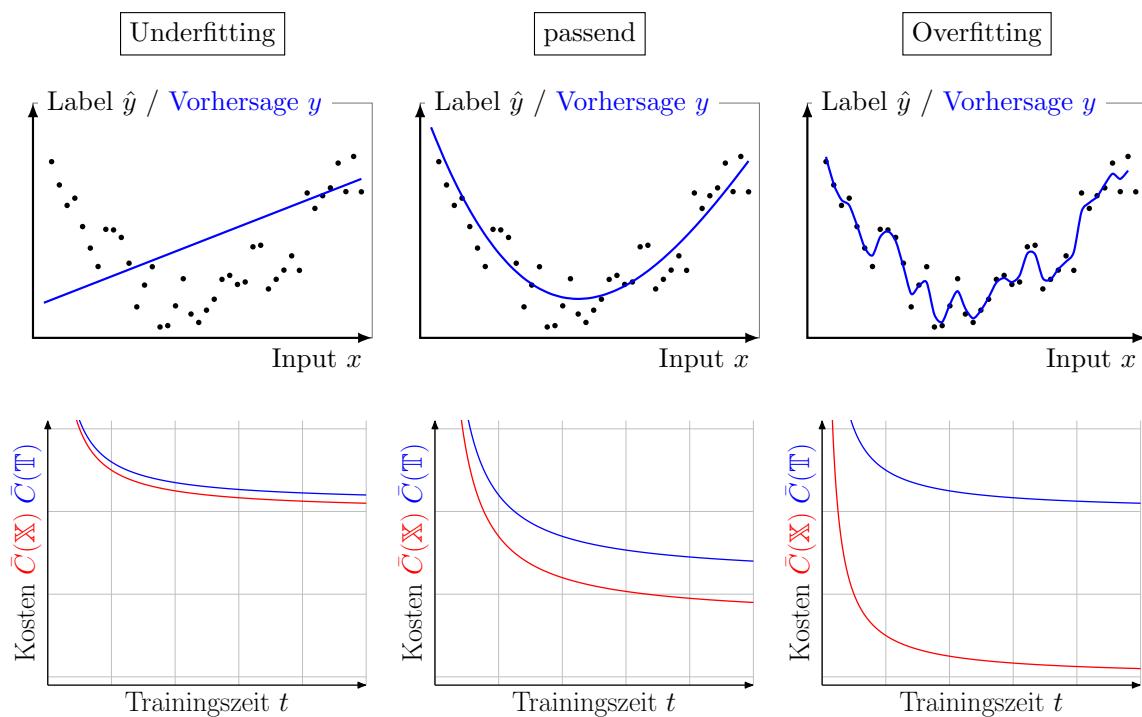


Abbildung 1.1: Visualisierung von Under- und Overfitting

Kapitel 2

Deep Learning und Künstliche Neuronale Netze

Nach dem im ersten Kapitel die Basis für ein fundiertes Verständnis von Maschinellem Lernen gelegt worden ist, soll es nun im zweiten Kapitel darum gehen, ein spezifisches Modell, namentlich Künstliche Neuronale Netze, zu erläutern. Darauf aufbauend wird in Kapitel drei die spezifische Architektur eines Künstlichen Neuronalen Netzes, namentlich das Convolutional Neural Network, dargelegt.

Die wohl besten Resultate für die meisten Problemstellungen des Maschinellen Lernens (Bilderkennung, Spracherkennung, etc.) werden durch **Künstliche Neuronale Netze** (engl.: neural networks, kurz: KNN) geliefert. Man bezeichnet diesen Bereich des Maschinellen Lernens auch als **Deep Learning**.

Künstliche Neuronale Netze sind vor allem biologisch durch Nervensysteme von Lebewesen inspiriert. Sie sind aber lediglich eine Abstraktion dieser Informationsverarbeitung und versuchen nicht eine möglichst genaue biologische Abbildung darzustellen. Es gibt nicht nur eine Art von Neuronalem Netz, sondern es existieren die verschiedensten Architekturen, welche je nach Problemstellung ausgewählt werden müssen. Diese Arbeit wird vor allem von zwei solcher Architekturen Gebrauch machen: Convolutional Neural Networks und sogenannte Autoencodern.

Quellen: (2)

2.1 Perzeptron

Um den Aufbau und die Funktion eines Künstlichen Neuronalen Netzes besser zu verstehen, wird im folgenden ein Vorgänger des KNN erklärt: das **Perzeptron**.

Das einlagige Perzeptron wurde erstmals 1958 von Frank Rosenblatt vorgestellt. Dieses besteht aus einem einzigen Künstlichen Neuron. Dieses künstliche Neuron hat mehrere binäre Inputs und einen einzigen binären Output. Binär bedeutet, dass der Wert nur entweder 0 (*aus*) oder 1 (*ein*) sein kann. Des Weiteren besitzt es mehrere sogenannte **Gewichte** $w_1, \dots, w_m \in \mathbb{R}$, für jeden Input x_i ein Gewicht w_i . Diese sind reelle Zahlen, welche das Verhalten des Perzeptron bestimmen. Die **gewichtete Summe**, also die Summe aller Produkte der Inputs mit ihrem Gewicht, wird mit \tilde{z} bezeichnet. Sie ist das gleiche wie das Skalarprodukt des Gewichtvektors $\mathbf{w} = (w_1 \ \dots \ w_m)^\top$ mit dem Inputvektor \mathbf{x} .

$$\tilde{z} = \sum_{i=1}^m w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

Zusätzlich besitzt das Perzeptron einen **Schwellenwert** \tilde{b} . Zusammen mit den Gewichten bilden sie die Modellparameter. Das Perzeptron verhält sich so, dass falls die gewichtete Summe \tilde{z} grösser als der Schwellenwert \tilde{b} ist, das Neuron feuert. Das bedeutet der Output beträgt 1; andernfalls ist er 0 (siehe erster Teil der Hypothesenfunktion h in Gleichung (2.1)). Es ist gängig, die Ungleichung der Bedingung in die Nullstellenform zu bringen und \tilde{b} durch die **Neigung** (engl.: bias) $b = -\tilde{b}$ zu ersetzen. Somit lautet die Ungleichung: $\tilde{z} + b > 0$. Der neue Term $\tilde{z} + b$ wird mit z bezeichnet (siehe Rest der Gl. (2.1)). Die Neigung gibt an, wie stark das Neuron dazu neigt, zu feuern.

$$h(\mathbf{x}) = \begin{cases} 1 & \text{falls } \tilde{z} > \tilde{b} \\ 0 & \text{ansonsten} \end{cases} = \begin{cases} 1 & \text{falls } z > 0 \\ 0 & \text{ansonsten} \end{cases} = \begin{cases} 1 & \text{falls } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{ansonsten} \end{cases} \quad (2.1)$$

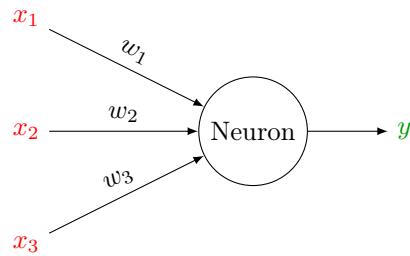


Abbildung 2.1: Perzepron mit drei Inputs

Für das Trainieren des Perzeprons existieren spezielle Verfahren, welche hier aber nicht weiter beleuchtet werden sollen. Dies aus dem Grund, weil das Gradientenverfahren hier nicht verwendet werden kann. Der Grund dafür soll wird in Sektion (2.2.2) erläutert.

Quellen: (5) (3) (2)

2.1.1 Lernpotenzial eines Perzeprons

Nun stellt sich die Frage, was ein Perzepron eigentlich erlernen kann und wofür es nutzbar ist. Das Perzepron ist lediglich ein **linearer Klassifikator** der Form $y = w_1x_1 + \dots + w_mx_m$. Es ist also ein Klassifizierungsmodell und kein Regressionsmodell. Es kann die Features in zwei Klassen 0 oder 1 einordnen, wobei der Output der Hypothesenfunktion diese Klassifizierung angibt. Überschreitet y den Schwellenwert \tilde{b} , werden die Features der Klasse 1 zugeordnet, sonst der Klasse 0. Jedoch müssen diese Klassen linear separierbar sein.

Lineare Separierbarkeit bedeutet, dass alle Featurevektoren $\mathbf{x}_1, \dots, \mathbf{x}_p \in \mathbb{R}^m$ innerhalb ihres Vektorraums \mathbb{R}^m durch eine Hyperebene in ihre Klassen aufteilbar sein müssen. Falls das Perzepron zwei Inputs besitzt, bedeutet dies, dass die Ortsvektoren durch eine Gerade voneinander trennbar sein müssen (siehe Abb. (2.2)).

Falls die Features nicht linear separierbar sind, kann das Perzepron die Klassifizierung nicht erlernen. Somit ist diese Modell ziemlich primitiv und kann nicht auf komplizierte nicht-lineare Problemstellung angewandt werden.

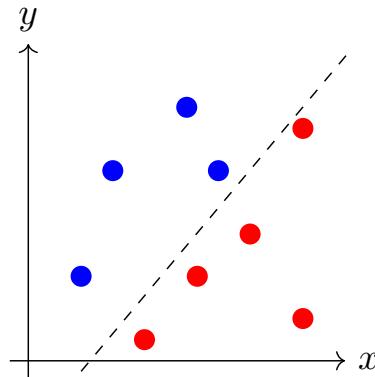


Abbildung 2.2: zwei-dimensionale lineare Separierung

Quellen: (5) (6)

2.2 Erweiterung der künstlichen Neuronen

Ein Perzepron ist, wie vorhin erklärt, nur in der Lage lineare Klassifikationen durchzuführen. Um nun auch Regressionsprobleme zu lösen, muss das Konzept des künstlichen Neurons ausgebaut werden. Wir benötigen ein Neuron, welches sich besonders gut als Baustein für KNNs eignet.

2.2.1 Künstliche Neuronen im Allgemeinen

Künstliche Neuronen sind immer so aufgebaut, dass sie einen oder mehrere Inputs und nur einen einzigen Output besitzen. Zu jedem Input x_i ist ein Gewicht w_i assoziiert. Zuerst wird die gewichtete Summe der Inputs \tilde{z}

gebildet. Die Neigung b wird ebenfalls dazu addiert, um z zu erhalten. Nun muss die sogenannte **Aktivierung** a gebildet werden. Sie ist der Output des Neurons. Die Aktivierung $a = \varphi(z)$ ist das Resultat der **Aktivierungsfunktion** $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ angewendet auf z . Die verschiedenen künstlichen Neuronen unterscheiden sich fast nur in ihrer Aktivierungsfunktion.

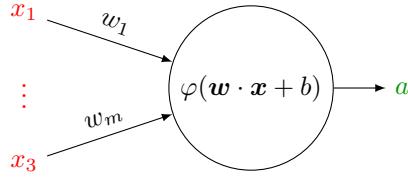


Abbildung 2.3: ein künstliches Neuron

Quellen: (3) (7) (2)

2.2.2 Perzeptronen als künstliche Neuronen

Nun nochmal ein Blick auf das Perzepron im Angesicht der Aktivierungsfunktion. Ein wesentlicher Unterschied des Perzeprons gegenüber sonstigen künstlichen Neuronen besteht darin, dass seine Inputs und Outputs nur binäre Werte annehmen können. Um dieses Verhalten des Perzeprons zu erhalten, muss eine Stufenfunktion als Aktivierungsfunktion verwendet werden: die Heaviside-Funktion Θ . Sie hat einen einzigen Stufensprung bei $x = 0$ vom Wert 0 auf 1 (siehe Abb. 2.4). Eine Konsequenz dieser Stufenfunktion ist, dass das Gradientenverfahren hier nicht angewendet werden kann, da die Ableitung der Heaviside-Funktion entweder nicht definiert ist ($x = 0$) oder überall 0 beträgt.

$$\varphi^{\text{hlim}}(z) = \Theta(z) = \begin{cases} 1 & \text{falls } z \geq 0 \\ 0 & \text{falls } z < 0 \end{cases}$$

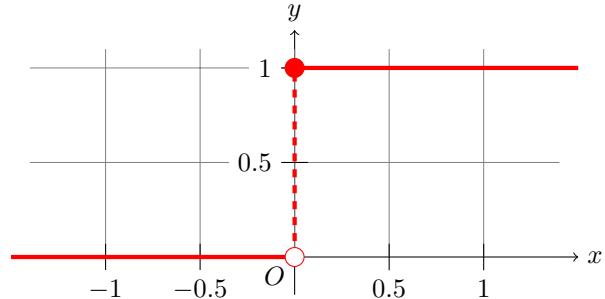


Abbildung 2.4: Definition und Graph der Heaviside-Funktion Θ

Quellen: (7) (5) (2)

2.2.3 ReLU-Neuronen

Der nächste Schritt nach einer Stufenfunktion als Aktivierungsfunktion, sind lineare Aktivierungsfunktionen. Für diese können die Inputs nun beliebige reelle Zahlen sein. Jedoch sind solche linearen Neuronen in einem KNN von keinerlei Nutzen. Dies ist dadurch begründet, dass eine Verkettung von linearen Neuronen immer auf eine einzige lineare Funktion reduziert werden kann. Somit hat die Verkettung keinen Mehrwert.

Stattdessen verwendet man sogenannte ReLU-Neuronen. Sie benutzen die **Rectified Linear Unit (ReLU)** als Aktivierungsfunktion. Diese ist eine nur teilweise lineare Aktivierungsfunktion. Die Werte grösser als 0 werden auf sich selbst linear abgebildet und die Werte kleiner oder gleich 0 werden auf 0 abgebildet (siehe Abb. 2.5). Eine sehr wichtige Eigenschaft der ReLU-Funktion ist, dass sie - im Gegensatz zu der vorherigen Heaviside-Funktion Θ - fast überall differenzierbar¹ und monoton steigend ist. Erst für diese Aktivierungsfunktion kann das Gradientenverfahren angewendet und somit das KNN trainiert werden.

Da die ReLU-Funktion nur teilweise linear ist, gehört sie genau genommen den nicht-linearen Aktivierungsfunktionen an. Diese Nicht-Linearität erlaubt es dem Neuron, deutlich komplexere Systeme zu modellieren bzw.

¹Eigentlich ist die ReLU-Funktion in $x = 0$ wegen des Knicks nicht differenzierbar. Für die Gradientenberechnung definiert man jedoch einfach die Ableitung $\varphi'^{\text{ReLU}}(0) := 0$. Dies ist mathematisch zwar nicht korrekt, löst aber das Problem.

deutlich komplexere Probleme zu lösen. In Sektion (2.7) wird dargelegt, dass eine Komposition von nicht-linearen Neuronen jede beliebige Funktion approximieren kann.

Das ReLU-Neuron werden wir an dieser Stelle zur Seite legen und erst wieder in Kapitel (3) im Zusammenhang mit KNNs zur Bilderkennung betrachten.

$$\varphi^{\text{ReLU}}(z) = \begin{cases} z & \text{falls } z > 0 \\ 0 & \text{falls } z \leq 0 \end{cases} = \max(z, 0) \quad (2.2)$$

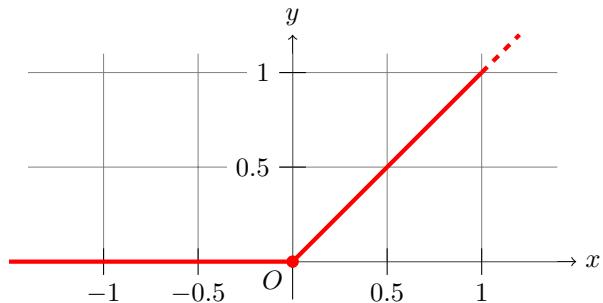


Abbildung 2.5: Formel und Graph der ReLU-Funktion

Quellen: (7) (3) (2)

2.2.4 Sigmoid-Neuronen

Ein weiteres nicht-lineares Neuron sind sogenannte Sigmoid-Neuronen. Die Bezeichnung stammt von ihrer Aktivierungsfunktion: der Sigmoidfunktion σ . Sie sind die meist verwendeten Neuronen in klassischen KNNs. Auch sie können aufgrund ihrer Nicht-Linearität zur Approximation jeder Funktion verwendet werden. Sie weicht stark von der Linearität ab und ist somit auch in der Lage komplexe Sachverhalte zu modellieren.

Die Sigmoid-Funktion besitzt eine einzige Wendestelle $\sigma''(x = 0) = 0$ und hat zwei Asymptoten, eine $\lim_{x \rightarrow -\infty} \sigma(x) = 0$ und eine zweite $\lim_{x \rightarrow \infty} \sigma(x) = 1$ (siehe Abb. (2.6)). Des Weiteren zeichnet sie sich durch eine vergleichsweise einfache Ableitung aus.

$$\varphi^{\text{sig}}(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

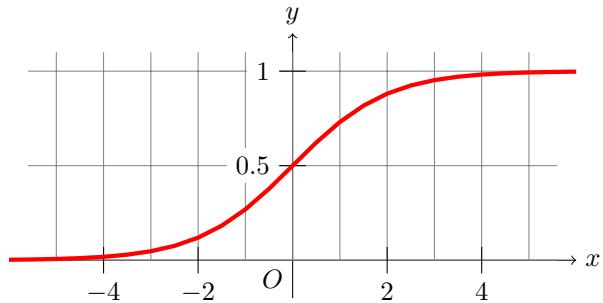


Abbildung 2.6: Definition, Ableitung und Graph der Sigmoid-Funktion σ

Quellen: (7) (8) (2)

2.3 Topologie der Künstlichen Neuronalen Netze

Nun sollen die Sigmoid-Neuronen als Bausteine für Künstliche Neuronale Netze Verwendung finden. Dazu werden sie miteinander verbunden und bilden so ein Netz, ähnlich wie ein Nervensystem.

Diese Neuronen sind in verschiedenen Schichten (engl.: layers) arrangiert. Die erste bildet die **Inputschicht**. Sie beinhaltet die Inputneuronen. Diese sind eigentlich keine richtigen Neuronen, sondern eher Platzhalter für ihr jeweiliges Feature x_i . Als Letztes kommt die **Outputschicht** mit den Outputneuronen, welche jeweils einen Outputwert y_i besitzen. Dazwischen liegen die **Zwischenschichten** (engl.: hidden layers). Von ihnen kann es beliebig viele geben, und in ihnen können beliebig viele Neuronen liegen. Falls viele Zwischenschichten verwendet werden, bezeichnet man das Netzwerk als “deep”. Daher röhrt auch der Begriff des Deep Learning. Den Aufbau eines KNN bezeichnet man als **Topologie** des Netzes. Die Topologie umfasst viele Hyperparameter. Darunter sind zum Beispiel die Anzahl der Zwischenschichten, wie auch die Anzahl der Neuronen pro Schicht.

Jedes Neuron aus einer Schicht ist mit jedem Neuron aus der nächsten Schicht über Verbindungen gekoppelt. Alle Verbindungen besitzen ein Gewicht analog zu den Inputs des Perzeptrons. Die Aktivierung, also der Output, eines Neurons wandert entlang den jeweiligen Verbindungen zu allen Neuronen der nächsten Schicht und dienen als deren Input.

In Abbildung (2.7) ist ein Beispiel eines Neuronalen Netzes abgebildet. In diesem Fall besitzt es sowohl 4 Inputs, als auch 4 Outputs. Es hat ausserdem 3 Zwischenschichten. Die Erste und die Dritte haben jeweils 3 Neuronen und die Zweite besitzt 4 Neuronen.

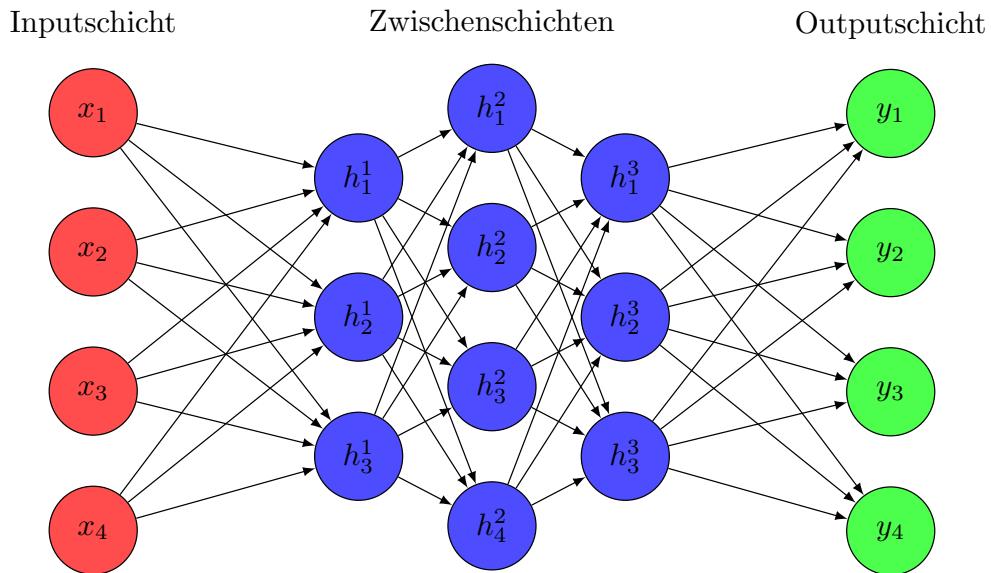


Abbildung 2.7: Schichten eines KNNs

Quellen: (9) (3) (2)

2.4 Lernverhalten

Die Hoffnung beim Trainieren von KNNs besteht darin, dass das Modell für jede weitere Schicht ein höheres Abstraktionsniveau erreicht. Würde man zum Beispiel ein Netzwerk zur Gesichtserkennung trainieren, könnte man sich den Erkennungsprozess folgendermassen vorstellen: Die erste Zwischenschicht erkennt Kanten und Konturen. Die zweite vereint diese Merkmale zu Ecken und primitiven geometrischen Formen. Die dritte Schichte sollte dann schon komplexere geometrische Formen erkennen, welche gewissen Gesichtsmerkmalen, wie der Nase, ähneln. Die letzte Schichte soll dann alle diese Merkmale zusammensetzen und so ein Gesicht als Ganzes erkennen.

2.5 Vorwärtspropagierung

Jetzt, da der Aufbau eines KNNs erläutert wurde, soll die mathematische Funktionsweise des Modells erklärt werden. Der Prozess der Berechnung der Outputwerte wird **Vorwärtspropagierung** genannt. Dieses Verfahren gibt dieser Art von KNN auch den Namen: **feedforward neural network**. Für das Verständnis müssen einige Konventionen zur Bezeichnung der Teile eines KNNs getroffen werden. Es sollten zusätzlich noch Abbildungen (2.8) und (2.9) zum besseren Verständnis der Nomenklatur studiert werden.

- l ist der Index einer Schicht. Die Indexierung beginnt bei der Inputschicht mit 0.
- L ist der letzte Schichtindex und somit auch die gesamte Anzahl an Schichten (ohne die Inputschicht).
- $|l|$ ist die Anzahl Neuronen in der l -ten Schicht².
- n_j^l bezeichnet das j -te Neuron in der l -ten Schicht.
- z_j^l ist die gewichtete Summe der Inputs des j -ten Neuron in der l -ten Schicht.

²Diese Schreibweise hat nichts mit dem Betrag zu tun, sondern wird gewählt, da sie sehr platzsparend ist.

- a_j^l ist die Aktivierung (bzw. der Output) des j -ten Neurons in der l -ten Schicht.
- b_j^l ist die Neigung für das j -te Neuron in der $(l + 1)$ -ten Schicht³.
- $w_{j,k}^l$ ist das Gewicht der Verbindung vom k -ten Neuron in der l -ten Schicht zum j -ten Neuron in der $(l + 1)$ -ten Schicht⁴.
- φ ist die gewählte Aktivierungsfunktion. Diese ist immer eine nicht-lineare Aktivierungsfunktion.

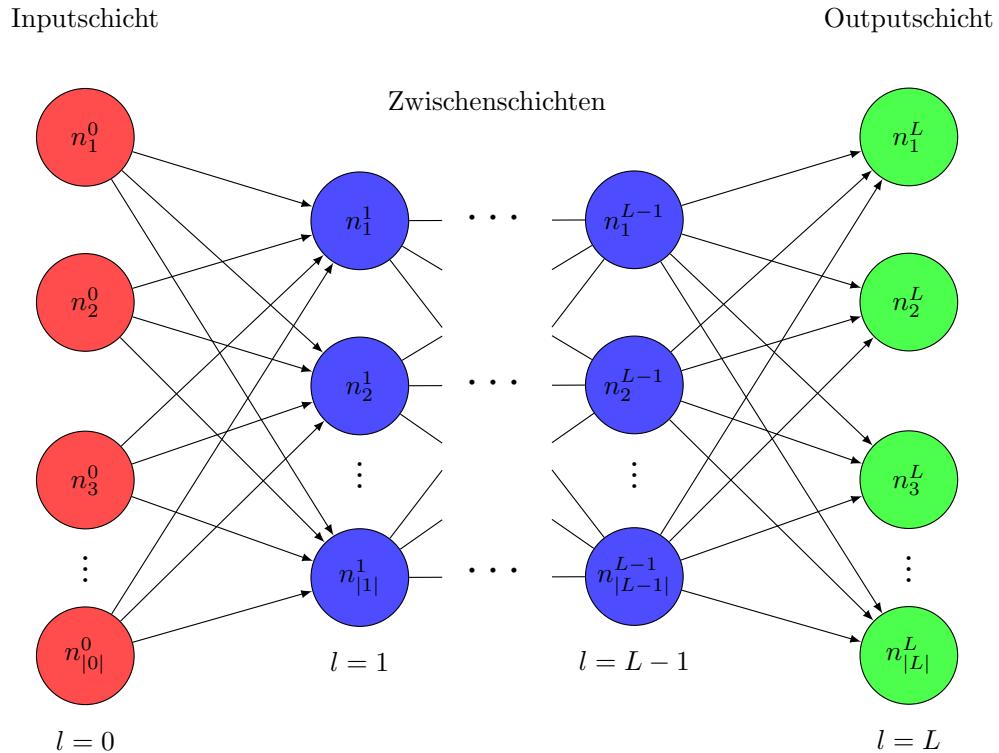


Abbildung 2.8: zum Verständnis der Nomenklatur der Neuronen

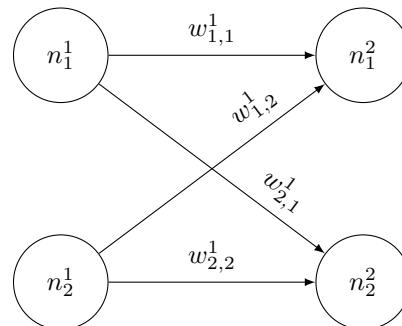


Abbildung 2.9: zum Verständnis der Gewichtebezeichnungen

Die Vorwärtspropagierung beginnt bei den Inputneuronen, welche jeweils einen Inputwert in sich tragen. Diese Werte werden, um für eine kohärente Nomenklatur zu sorgen, analog zu den Aktivierungen der anderen Neuronen mit a_j^0 bezeichnet, wobei j der Index des Neurons ist.

Nun müssen die restlichen Aktivierungen der Neuronen bis und mit den Outputneuronen berechnet werden. Dies geschieht rekursiv, anhand der Aktivierungen der vorherigen Schicht, und zwar folgendermassen (ersichtlich in Gleichung (FP1)):

Zuerst läuft eine Summe über alle Neuronen n_k^l der jetzigen Schicht l . Dabei wird die gewichtete Summe der

³Diese Konvention wurde gewählt, damit die folgenden Gleichungen simpler sind.

⁴Man beachte die Reihenfolge der Indizes!

Diese Konvention scheint zwar auf den ersten Blick unintuitiv, macht jedoch Sinn für die Matrixindexierung. (2.6).

Aktivierungen a_k^l mit den assoziierten Gewichten $w_{j,k}^l$ gebildet. Hierbei ist das Gewicht jenes, welches das k -te Neuron der l -ten Schicht mit dem j -ten Neuron der $(l+1)$ -ten Schicht verbindet (siehe Abb. (2.9)). Zusätzlich gehört zu der gewichteten Summe auch die jeweilige Neigung b_j^l , welche dazu addiert wird. Diese gewichtete Summe wird mit z_j^{l+1} bezeichnet.

$$z_j^{l+1} = \sum_{k=1}^{|l|} w_{j,k}^l a_k^l + b_j^l \quad (\text{FP1})$$

Auf diese Summe wird dann die Aktivierungsfunktion φ angewandt. Das ist dann die Aktivierung a_j^{l+1} des j -ten Neurons in der $(l+1)$ -ten Schicht.

$$a_j^{l+1} = \varphi \left(\sum_{k=1}^{|l|} w_{j,k}^l a_k^l + b_j^l \right) = \varphi(z_j^{l+1}) \quad (\text{FP2})$$

Für Deep Learning braucht man vor allem sogenannte Deep Neural Networks. Diese zeichnen sich dadurch aus, dass sie sehr viele Zwischenschichten besitzen. Deshalb bezeichnet man sie als “deep”. Bei solchen Netzwerken ist es nicht unüblich, dass sie sehr viele Neuronen und Verbindungen (über 100'000) besitzen. Um hier nicht den Überblick zu verlieren bzw. damit nicht zu viele Indizes notwendig sind, macht man Gebrauch von **Linearer Algebra**. Man verwendet Matrizen und Vektoren, um die vielen Variablen zusammenzufassen. Außerdem besteht ein weiterer Vorteil darin, dass Computer mithilfe von Vektor- und Matrixoperationen die Berechnungen parallelisieren können und in kürzerer Zeit und mit weniger Ressourcen viele Berechnungen gleichzeitig ausführen können. Dies beschleunigt das Training der Modelle um ein Vielfaches. In Anhang (C) wird dies thematisiert.

Die Inputs \mathbf{x} , Vorhersagen \mathbf{y} und Labels $\hat{\mathbf{y}}$ wurden bereits zu Beginn als Vektoren geschrieben. Nun sollen noch die Modellparameter und die restlichen Komponenten eines KNNs als Vektoren und Matrizen zusammengefasst werden. Sowohl alle gewichteten Summen z_j^l , wie auch alle Aktivierungen a_j^l einer Schicht l , werden in Vektoren $\mathbf{z}^l \in \mathbb{R}^{|l|}$ und $\mathbf{a}^l \in \mathbb{R}^{|l|}$ zusammengefasst. Auch alle Neigungen b_j^l für eine Schicht $(l+1)$ bilden einen Vektor $\mathbf{b}^l \in \mathbb{R}^{|l+1|}$.

Zu guter Letzt, wird noch eine **Gewichtsmatrix** $\mathbf{W}^l \in \mathbb{R}^{|l+1| \times |l|}$ definiert. Sie enthält alle Gewichte, welche die l -te Schicht zu der $(l+1)$ -ten Schicht verbindet. Das heisst, der Eintrag in der j -ten Zeile und in der k -ten Spalte ist $w_{j,k}^l$ und verbindet so das Neuron n_k^l zu dem Neuron n_j^{l+1} .

$$\begin{aligned} \mathbf{z}^l &= (z_1^l \ z_2^l \ \dots \ z_{|l|}^l)^T \\ \mathbf{a}^l &= (a_1^l \ a_2^l \ \dots \ a_{|l|}^l)^T \\ \mathbf{b}^l &= (b_1^l \ b_2^l \ \dots \ b_{|l+1|}^l)^T \\ \mathbf{W}^l &= \begin{pmatrix} w_{1,1}^l & w_{1,2}^l & \dots & w_{1,|l|}^l \\ w_{2,1}^l & w_{2,2}^l & \dots & w_{2,|l|}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{|l+1|,1}^l & w_{|l+1|,2}^l & \dots & w_{|l+1|,|l|}^l \end{pmatrix} \end{aligned}$$

Mit diesen Definitionen kann Gleichung (FP1) in Matrixform geschrieben werden. Dies, weil die Matrixmultiplikation von \mathbf{W}^l mit \mathbf{a}^l einen Vektor $\tilde{\mathbf{z}}^{l+1}$ ergibt, welcher alle gewichteten Summen \tilde{z}_j^{l+1} ohne die jeweilige Neigung enthält.

$$\mathbf{W}^l \mathbf{a}^l = \left(\sum_{j=1}^{|l|} w_{1,j}^l a_j^l \quad \sum_{j=1}^{|l|} w_{2,j}^l a_j^l \quad \dots \quad \sum_{j=1}^{|l|} w_{|l+1|,j}^l a_j^l \right)^T = \tilde{\mathbf{z}}^{l+1}$$

Nun muss noch der Neigungsvektor \mathbf{b}^l dazu addiert werden, damit die Gleichung (FP1a) entsteht, mit welcher der Vektor der gewichteten Summen \mathbf{z}^{l+1} gebildet werden kann.

$$\mathbf{z}^{l+1} = \mathbf{W}^l \mathbf{a}^l + \mathbf{b}^l \quad (\text{FP1a})$$

Im letzten Schritt wird die Aktivierungsfunktion auf \mathbf{z}^{l+1} angewendet, um den Aktivierungsvektor \mathbf{a}^{l+1} zu bilden. Hierfür muss aber noch ein neues mathematisches Konzept eingeführt werden: die Vektorisierung einer Funktion.

Vektorisierung einer Funktion

Die Vektorisierung einer skalaren Funktion f , geschrieben als $\mathbf{f}[\mathbf{v}]$ hat als Argument einen Vektor \mathbf{v} , auf dessen Komponenten jeweils *einzel*n die Funktion f angewendet wird. Dieser neue Vektor ist der Rückgabewert der Funktion. Er besitzt die gleichen Dimensionen wie der Argumentvektor.

$$\mathbf{f}[\mathbf{v}] = \begin{pmatrix} f(v_1) \\ \vdots \\ f(v_n) \end{pmatrix}$$

Nun kann die vektorisierte Aktivierungsfunktion φ auf \mathbf{z}^{l+1} angewendet werden.

$$\mathbf{a}^{l+1} = \varphi [\mathbf{W}^l \mathbf{a}^l + \mathbf{b}^l] = \varphi [\mathbf{z}^{l+1}] \quad (\text{FP2a})$$

Quellen: (3)

2.5.1 Initialisierung der Modellparameter

Ein weiterer Schritt, der vollzogen werden muss, bevor das Training beginnen kann, ist das Initialisieren aller Modellparameter, in diesem Fall die Gewichte und Neigungen. Dies ist ein sehr essenzieller Schritt, da diese Initialwerte die Leistungsfähigkeit des Modells erheblich beeinflussen.

Wie in Sektion (1.2.2) gezeigt, muss am Anfang des Gradientenverfahrens ein Startpunkt $\boldsymbol{\lambda}_{t=0} = (\lambda_1 \dots \lambda_k)^\top$ innerhalb des Gradientenfeldes ∇C gewählt werden, von welchem aus der Gradientenabstieg beginnt. Dieser Startpunkt entscheidet darüber, in welches lokale Minimum konvergiert wird und bestimmt somit auch die bestmögliche Exaktheit der Vorhersagen. Falls schlechte Initialwerte gewählt wurden, konvergiert der Punkt in ein hohes lokales Minimum, was grosse Kostenfunktionswerte und schlechte Vorhersagen verursacht.

Es ist nicht möglich, im Vorhinein zu wissen, welche Initialwerte gute Resultate liefern. Es müssen verschiedene Werte ausprobiert werden. Dafür initialisiert man gängigerweise die Modellparameter mit Zufallswerten. Zu diesem Zweck werden nicht irgendwelche Zufallsvariablen verwendet, sondern es gelangt die Gauss'sche Normalverteilung $\mathcal{N}(\mu, \sigma^2)$ bzw. ihre Dichtefunktion (siehe Abb. (2.10)) zur Anwendung. Diese ist folgendermassen definiert:

$$\phi(x | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left\{ -\frac{(x - \mu)^2}{2\sigma^2} \right\}$$

In den Anfängen des Maschinellen Lernens wurde häufig die normierte Normalverteilung verwendet. Jedoch hatte dies zur Konsequenz, dass mit jeder Schicht die Outputs eine stetig grösser werdende Standardabweichung σ aufwiesen. In Kombination mit Sigmoid-Neuronen, kommt es zu einer Verlangsamung des Lernprozesses. Dieses Problem ist als das **Vanishing-Gradient-Problem** bekannt, auf welches hier nicht weiter eingegangen wird.

In der Konsequenz wurde eine neue Technik entwickelt: die **Glorot-Initialisierung** (auch Xavier-Initialisierung genannt). Dabei gelangt erneut eine Normalverteilung mit Erwartungswert $\mu = 0$ zur Anwendung. Die Standardabweichung σ wird anhand der Grösse einer Schicht skaliert. Für eine Schicht l wird der Durchschnitt zwischen der Anzahl Inputs und der Anzahl Neuronen einer Schicht berechnet $r = \frac{|l-1|+|l|}{2}$. Der Kehrwert davon ist die Varianz der Normalverteilung. Die Gewichte werden folgendermassen initialisiert:

$$w_{t=0}^l \sim \mathcal{N} \left(\mu = 0, \sigma^2 = \frac{2}{|l-1| + |l|} \right) \quad (2.3)$$

Durch dieses Verfahren bleibt die Varianz innerhalb einer Schicht erhalten. Teilweise wird dieses Verfahren auch für die Neigungen b angewendet. Eine andere Möglichkeit besteht in der Initialisierung der Neigungen mit 0, wie wir das machen werden.

$$b_{t=0}^l = 0 \quad (2.4)$$

Quellen: (10) (3) (2)

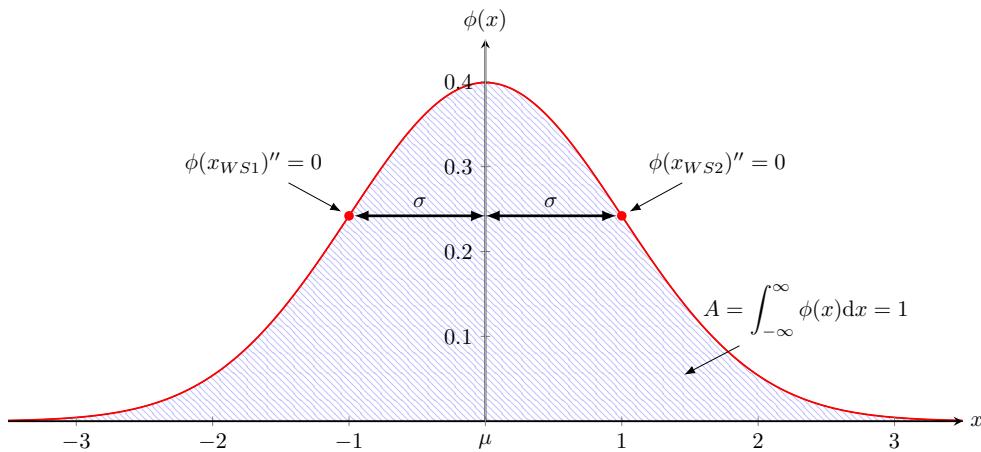


Abbildung 2.10: Graph der Dichtefunktion $\phi(x | \mu = 0, \sigma^2 = 1)$ mit ihren wichtigsten Eigenschaften

2.6 Rückwärtspropagierung

Ein KNN wird üblicherweise, wie die meisten Modelle, mithilfe des Stochastischen Gradientenverfahrens trainiert. Die wahre Herausforderung besteht darin, die partiellen Ableitungen der Kostenfunktion bezüglich der Modellparameter zu berechnen. Anders gesagt müssen alle Terme $\frac{\partial C}{\partial w_{j,k}^l}$, wie auch alle Terme $\frac{\partial C}{\partial b_k^l}$, bestimmt werden. Das Verfahren zur Ermittlung dieser Ausdrücke in KNNS ist so spezifisch und aufwendig, dass es einen eigenen Namen besitzt: die sogenannte **Rückwärtspropagierung** (engl.: backpropagation, auch Fehlerrückführung).

Für das Verständnis von ML ist es nicht essenziell, die Rückwärtspropagierung zu verstehen. Jedoch kann so nachvollzogen werden, wie SGD in der Praxis funktioniert. Grob umrissen besteht der Grundgedanke darin, die Werte der Kostenfunktion rückwärts durch das Modell zu schicken und dabei die partiellen Ableitungen mithilfe der Kettenregel zu bestimmen. Daher hat das Verfahren auch seinen Namen.

Für eine ausführliche Herleitung ist auf Anhang B zu verweisen, in welchem ebenfalls auf das Konzept eines Computational Graphs eingegangen wird.

2.7 Universal Approximation Theorem

Es stellt sich nun die Frage, was ein Neuronales Netz alles erlernen kann. Diese Frage kann mithilfe des **Universal Approximation Theorem** (UAT) beantwortet werden. Es handelt sich um einen mathematischen Beweis dafür, dass ein KNN grundsätzlich in der Lage ist, jede kontinuierliche Funktion beliebig exakt zu approximieren.

Etwas genauer ausgedrückt, besagt der UAT, dass ein KNN mit einer einzigen Zwischenschicht, welche eine endliche Anzahl Neuronen besitzt, sich jeder kontinuierlichen Funktion beliebig stark annähern kann. Voraussetzung dafür ist, dass es sich bei den Neuronen um nicht-lineare Neuronen handelt. Da sich die meisten Klassen von Problemen als eine Funktion formulieren lassen, bedeutet dies, dass ein KNN theoretisch jedes Problem lösen kann! Jedoch handelt es sich beim UAT nur um eine theoretische Aussage über das Lernpotenzial eines KNNS. Somit trifft das Theorem keinerlei Aussage darüber, ob ein KNN wirklich erfolgreich darin ist, die jeweilige Funktion zu erlernen. Ein mögliches Hindernis könnte in Overfitting bestehen.

Da der eigentliche Beweis mathematisch ziemlich anspruchsvoll ist, wird er im Rahmen dieser Arbeit nicht weiter behandelt.

Quellen: (3) (11)

Kapitel 3

Convolutional Neural Networks

Nachdem in Kapitel zwei das Wesen von Künstlichen Neuronalen Netzen charakterisiert wurde, soll es nun in Kapitel drei darum gehen, eine spezifische Architektur eines KNNs zu erklären, welche sich besonders gut für Bilderverarbeitung eignet. Es handelt sich um das Convolutional Neural Network.

Viele Anwendungen von Machine Learning sind mit einer Bild- oder Audioverarbeitung verbunden, wie z. B. Bildklassifizierung, Gesichts- oder Spracherkennung. Vor allem für hochauflösende Bilder sind die KNNs, wie wir sie soeben kennengelernt haben, jedoch nicht geeignet. Sie sind zum Teil gar nicht in der Lage, eine Korrelation zwischen den Inputs und Outputs zu erlernen. Um diesen Umstand zu erklären, wird ein kleines Beispielmodell erläutert:

Es soll ein KNN entworfen werden, welches eine Photographie danach klassifizieren soll, ob ein Hund darauf sichtbar ist oder nicht. Für dieses Gedankenexperiment wird ein relativ niedrig aufgelöstes Bild mit 256×256 Pixel gewählt (dies entspricht weniger als 0.07 Megapixel; im Vergleich dazu: Ein iPhone XS hat eine Kamera mit 12 Megapixel). Um die verschiedenen Farben zu codieren, besitzt jeder Pixel drei Farbkomponenten: R (rot), G (grün) und B (blau). Somit hat dieses Bild insgesamt $256 \times 256 \times 3 = 196'608$ Komponenten. Jede Komponente ist ein Feature, welches das KNN zu verarbeiten hat. So bestünde die erste Schicht des Netzwerkes aus fast 200'000 Neuronen. Um diese Schicht nun mit seiner Nachbarschicht zu verbinden, welche die gleiche Größe besitzt, werden $196'608 \times 196'608 = 38'654'705'664$ Verbindungen und damit gleich viele Gewichte benötigt! Für ein Netzwerk ohne eine einzige Zwischenschicht gäbe es also über 38 Milliarden Modellparameter zu erlernen! Dass dies nicht realistisch ist, liegt auf der Hand.

Nicht nur die Anzahl der Modellparameter ist ein Problem für KNNs in der Bildverarbeitung, sondern es existieren noch weiter Probleme. Ohne auf diese einzugehen, sollte nun klar sein, dass eine andere Modellarchitektur notwendig ist, um Machine Learning auf Bilder anwenden zu können. Für derartige Anwendungen wurde eine modifizierte Version eines KNNs entwickelt: das **Convolutional Neural Network** (CNN). Im Allgemeinen sind CNNs immer dann geeignet, wenn es Daten zu verarbeiten gilt, welche eine rasterartige Form aufweisen, wie z. B. Bilder. Diese Art von Netzwerk macht Gebrauch von Konzepten aus der klassischen Bildverarbeitung, wie sie beispielsweise auch Photoshop ermöglicht. Wie beim Perzepron und bei den klassischen KNNs wurde auch hier die Architektur von der Biologie inspiriert. Der folgende Abschnitt wird die Funktionsweise eines solchen CNNs erklären.

Quellen: (1) (12) (13)

3.1 Bilder als Tensoren

CNNs verarbeiten Bilder. Diese stellen den Input für die Modelle dar. Um diese Bilder erfassen zu können, ist es sinnvoll, sie als sogenannte **Tensoren** vom Rang 3 zu untersuchen, anstatt sie als Anordnungen von Pixeln zu betrachten. Um zu verstehen, was ein Tensor dritten Ranges ist, soll zunächst erläutert werden, was mit einem Tensor im Allgemeinen gemeint ist.

Tensor

Ein Tensor \mathbf{T} ist eine Verallgemeinerung von Skalaren, Vektoren und Matrizen auf n Dimensionen. Es handelt sich wie bei Matrizen um eine Zahlenanordnung. Dabei wird die Anzahl Dimensionen, innerhalb welcher die Zahlen liegen, als Rang oder Stufe n des Tensors bezeichnet. Vorstellen kann man sich einen Tensor als ein Hyperrechteck mit n Dimensionen, innerhalb dessen die Zahlen in einem Raster angeordnet sind. Diese Zahlen sind die Elemente des Tensors. Ein Tensor nullten Ranges ist ein Skalar, ein Tensor ersten Stufe ist ein Vektor und ein Tensor mit Rang 2 ist eine normale 2D-Matrix.

$$1 \in \mathbb{R} \text{ (Skalar)} \quad \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \in \mathbb{R}^3 \text{ (Vektor)} \quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \in \mathbb{R}^{2 \times 3} \text{ (Matrix)}$$

Tensor 3. Ranges

Ein Tensor $\mathbf{T} \in \mathbb{R}^{h \times w \times d}$ mit Rang 3 ist eine 3-dimensionale Zahlenanordnung. Man kann sich diesen Tensor als eine 3D-Matrix vorstellen; ein Volumen, innerhalb dessen die Elemente in einem Raster angeordnet sind. Analog zum Volumen bezeichnet man die Form des Tensors mit Höhe, Breite und Tiefe.

Die Schreibweisen-Konvention für Tensoren 3. Ordnung soll an folgendem Beispielstensor der Form $\mathbb{R}^{3 \times 3 \times 3}$ illustriert werden.

$$\mathbf{T} = \left[\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right] = \begin{array}{c} \text{A 3D matrix structure with dimensions 3x3x3, visualized as a cube with colored layers (blue, green, red) and highlighted elements (0, 1, -1).} \\ \left(\begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix} \right) \quad \left(\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \right) \quad \left(\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \right) \end{array} \in \mathbb{R}^{3 \times 3 \times 3}$$

Ein Bild kann somit als Tensor dritten Ranges $\mathbf{B} \in \mathbb{R}^{h \times w \times c}$ betrachtet werden, in der Form (Bildhöhe \times Bildbreite \times Anzahl Farbkomponenten). Die Elemente der Matrix nehmen dann die Werte der Pixelkomponenten an. Ein schwarz-weisses Bild hat nur eine Komponente, welche die Helligkeit angibt. Somit wäre es eine normale 2D-Matrix $\mathbf{B} \in \mathbb{R}^{h \times w}$.

Quellen: (12) (14)

3.2 Topologie der Convolutional Neural Networks

Ein CNN besteht, wie ein KNN auch, aus mehreren Schichten. Jede dieser Schichten erhält als Input ein Bild und hat auch wieder eines als Output. Ein wichtiger Unterschied des CNNs ist, dass es aus unterschiedlichen Typen von Schichten besteht. Grundsätzlich lassen sich vier Arten von Schichten unterscheiden:

- Fully-connected-Schichten
- Convolutional-Schichten
- Pooling-Schichten
- Upsampling-Schichten

Die Fully-Connected-Schicht ist bereits bekannt und verkörpert die klassische Schicht eines KNNs, bestehend aus Neuronen. Auf diese werden wir daher nicht weiter eingehen, da sie bereits im vorherigen Kapitel erläutert wurde.

Die Convolutional-Schicht ist eine neuartige Schicht, welche es im KNN nicht gibt. Sie ist die Schicht, welche für das Training relevant ist, da sie die Modellparameter beinhaltet. Sie extrahiert die relevanten Features aus den Inputbildern und lernt so die Bilder zu verstehen.

Die Pooling-Schichten, wie auch die Upsampling-Schichten, beinhalten keine Modellparameter und sind deshalb für das Training nicht direkt relevant. Diese Schichten werden lediglich benötigt, um die verarbeiteten Bilder neu zu skalieren. Dies ist sinnvoll, da durch die Extraktion gewisser Features die restlichen Features wegfallen. Somit muss weniger Information pro Bild gespeichert werden und die Bilder sollten schrumpfen, um Overfitting

vorzubeugen. Die Pooling-Schichten reduzieren die Bilder und die Upsampling-Schichten erweitern sie. Dies wird in Sektion ④ für die Entwicklung von Autoencoder hilfreich werden.
Grundsätzlich folgt auf eine Convolutional-Schicht entweder eine Pooling- oder eine Upsampling-Schicht. Somit bilden sie gewissermassen eine Einheit.

In Abbildung ③.1 ist ein Schema eines CNNs abgebildet.

Abbildung 3.1: Schichtung eines CNNs

Quellen: (1) (12) (13)

3.3 Convolutional-Schichten und Filter

Zuerst wird nun die Convolutional Schicht betrachtet. Diese Schicht macht ausgiebig Gebrauch von sogenannten Filtern. Diese werden in diesem Abschnitt ausführlich behandelt.

3.3.1 Filter in der Bildverarbeitung

Filter (auch Kerne) sind in der Bildverarbeitung sehr verbreitet. Jeder kennt sie entweder von Photoshop, von Instagram oder sonstigen Bildbearbeitungsprogrammen. Auch CNNs machen Gebrauch von solchen Filtern. In diesem Fall, um die Features eines Bildes zu erlernen.



Abbildung 3.2: der Instagramfilter ‘Amaro’ auf ein Beispielbild angewandt (15)

Ein Filter ist eine Region, die deutlich kleiner ist als das verarbeitete Bild \mathbf{B} , welche über alle Pixel wandert, diese manipuliert und so wieder ein neues Bild $\tilde{\mathbf{B}}$ erzeugt. Mathematisch gesehen handelt es sich bei einem solchen Filter um einen Tensor, den sogenannten **Filtertensor \mathbf{F}** oder auch Faltungstensor (siehe Abb. ③.3) und Abb. ③.7). Solche Filter sind immer quadratisch, wobei ihre Zeilen- und Spaltenlänge mit f bezeichnet wird. Dabei ist f immer eine ungerade Zahl, damit ein Element, das sogenannte **Zentralelement** (engl.: center element), bezeichnet mit $(\mathbf{F})_C$, immer im Zentrum des Filters liegt.

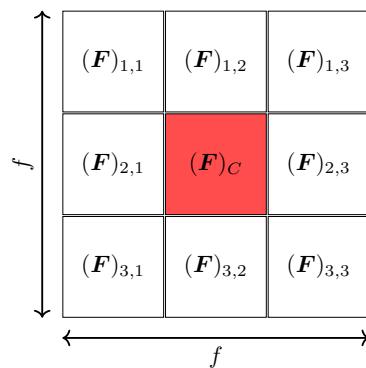


Abbildung 3.3: eine 2D-Filtermatrix mit rotem Zentralelement $(\mathbf{F})_C$

Das Verhalten des Filters wird durch seine Tensoreinträge bestimmt. Die Elemente können so gewählt werden, dass bei Anwendung des Filters auf ein Bild, dieser bestimmte Features hervorhebt. Solche Features könnten zum Beispiel Ränder oder Kanten sein, welche akzentuiert werden.

Nun wird ein beispielhafter Kantendetektionsfilter betrachtet. Er besitzt die Filtergrösse $f = 3$ und seine Einträge lauten folgendermassen:

$$\mathbf{F}_K = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix}$$

Angewendet auf ein schwarz-weisses Bild wird ein neues Bild erzeugt, bei welchem alle erkannten Kanten weiss eingefärbt werden und die restlichen Bereiche schwarz sind. Der Filter erkennt dabei jede Stelle als Kante, welche zwei Regionen mit genug grossem Kontrast voneinander trennt. In Abbildung (3.4) wurde der erwähnte Kantendetektionsfilter auf ein Beispielbild angewendet. Links ist das Ursprungsbild und rechts ist das generierte Bild zu sehen¹.



Abbildung 3.4: Kantendetektionsfilter angewandt auf Beispieldbild (15)

Quellen: (12) (16) (17)

3.3.2 Filter in CNNs

Aus didaktischen Gründen betrachten wird nun zunächst nur 2D-Filter, welche sich für graustufige Bilder $\mathbf{B} \in \mathbb{R}^{h \times w}$ eignen. Somit ist der hierfür ausgewählte Filter eine 2D-Matrix, bezeichnet mit $\mathbf{F} \in \mathbb{R}^{f \times f}$. Man bedenke, dass es sich im allgemeinen Fall bei jeder Filtermatrix um einen 3D-Tensor handelt.

Wie bereits dargelegt können Filter genutzt werden, um gewisse Features eines Bildes hervorzuheben und andere Features zu ignorieren. Dabei bestimmen die Filtereinträge, welche Features extrahiert werden. Somit besteht die Aufgabe darin, die richtigen Filtereinträge zu bestimmen, damit die gewünschten Features erkannt werden und auf den gewünschten Output abgebildet werden. Naheliegend sollte jetzt der Gedanke sein, die Filtereinträge als die Modellparameter eines CNNs zu definieren. Mithilfe von SGD sollen erneut diese Parameter erlernt werden.

Da die Filter die gleiche Funktion haben, wie die Gewichte in einem KNN, bezeichnet man die Filter in einem CNN mit \mathbf{W} . Die Grösse des Filters stellt dabei einen Hyperparameter dar.

$$\mathbf{W} = \begin{pmatrix} w_{1,1} & \cdots & w_{1,f} \\ \vdots & \ddots & \vdots \\ w_{f,1} & \cdots & w_{f,f} \end{pmatrix}$$

Somit haben wir erfolgreich das Problem mit der Überzahl an Modellparametern aus Sektion (3) behoben. Es existieren pro Sicht nur noch $f \times f$ Modellparameter, wobei f oft eine kleine Zahl ist.

3.3.3 Filteroperation intuitiv

Wie wendet man nun einen solchen Filter auf ein Bild an? Um das Prozedere einer Filteroperation leichter verständlich zu machen, soll es erneut für ein graustufiges Bild $\mathbf{B} \in \mathbb{R}^{h \times w}$ erläutert werden. Zu einem späteren Zeitpunkt wird auch die Anwendung auf farbige Bilder erläutert (siehe Sektion (3.3.4)).

Bei einer Filteroperation wendet man einen Filter \mathbf{W} auf eine Bildmatrix \mathbf{B} an und erhält so ein neues Bild $\tilde{\mathbf{B}}$. Man schreibt dafür: $\tilde{\mathbf{B}} = \mathbf{W} * \mathbf{B}$.

¹Dieses Bild wurde mithilfe von GIMP (GNU Image Manipulation Program) erzeugt. Dieses Programm bietet die Möglichkeit, eine Filtermatrix auf ein Bild anzuwenden.

Um nun diese Operation zu erklären, wird als Beispiel eine 2D-Filtermatrix $\mathbf{W} \in \mathbb{R}^{3 \times 3}$ verwendet, bei welcher $f = 3$ gewählt wurde. Wie bereits oben erwähnt, wandert der Filter über das Bild. Dabei befindet sich der Filter immer über einer Region des Bildes, welche gleich gross ist wie der Filter selbst (in diesem Fall (3×3)). Diese Region des Bildes bezeichnet man als das **Rezeptives Feld** des Filters (siehe Abb. (3.5)). Man bezeichnet es mit $\tilde{\mathbf{B}} \in \mathbb{R}^{3 \times 3}$. Somit ist das Rezeptive Feld eine Untermatrix der Obermatrix \mathbf{B} .

In diesem Bereich entsprechen den Filtermatrixeinträgen immer eindeutige Bildmatrixeinträge. Jedem Element des Filters entspricht ein Element des Rezeptiven Feldes. Dem Zentralelement $(\mathbf{W})_C$ ist dabei gerade der sogenannte Quellenpixel (engl.: source pixel) $(\mathbf{B})_S$ des Bildes zugeordnet. Ihn verwenden wir für eine Namenskonvention für das Rezeptive Feld. Das Rezeptive Feld wird mit $\hat{\mathbf{B}}^{(y,x)}$ bezeichnet, wobei die hochgestellten Indizes (y, x) die Position des Quellenpixel $(\mathbf{B})_S$ im Bild \mathbf{B} angibt.

Abbildung 3.5: ein Filter und sein rezeptives Feld

Der Filter beginnt nun oben links über das Bild zu wandern. Somit hat der Filter zu anfang das Rezeptive Feld $\tilde{\mathbf{B}}^{(2,2)}$, denn der Filter darf nicht über die Ränder des Bildes hinausragen. Nun werden die Elemente des Filters mit den Elementen des Rezeptiven Feldes verrechnet, welche die gleichen Dimensionen besitzen. Dabei wird das Hadamard-Produkt (das elementweise Produkt, erwähnt in Anhang (B)) der beiden Matrizen miteinander gebildet. Daraus resultiert eine neue Matrix $\mathbf{W} \odot \tilde{\mathbf{B}}^{(2,2)} = \mathbf{P} \in \mathbb{R}^{3 \times 3}$. Anschliessend werden alle Elemente der neuen Matrix \mathbf{P} aufsummiert. Dieser Wert ist dann der Grauwert des ersten Pixels $(\tilde{\mathbf{B}})_{1,1}$ des neu entstandenen Bildes (siehe Gl. (3.1)). In diesem Sinne stellt der Filter eine Gewichtung der Nachbarspixel des Ursprungsbild dar, welche bestimmt, wie die neuen Pixel aussehen.

$$(\tilde{\mathbf{B}})_{1,1} = \Sigma(\mathbf{P}) = \sum_{y=1}^3 \sum_{x=1}^3 (\mathbf{P})_{y,x} \quad (3.1)$$

Nun wird der Filter um ein Element nach rechts verschoben und die gleiche Prozedur angewendet, um den zweiten Pixel zu berechnen. Dies wird so lange vollzogen, bis eine ganze Zeile der Bildmatrix durchstreift wurde. Danach wird der Filter wieder ganz nach links verschoben und er bewegt sich ein Element nach unten. Das geschieht, bis das ganze Bild verrechnet wurde. Dabei werden die Elemente des ursprünglichen Bildes durchaus mehrfach verrechnet, da es zu einer Überlappung der vorherigen Position des Filters und der verschobenen Position kommt.

Es ist zu bemerken, dass das neue Bild $\tilde{\mathbf{B}} \in \mathbb{R}^{(h-2) \times (w-2)}$ nicht mehr die gleichen Massen aufweist wie das Ursprungsbild $\mathbf{B} \in \mathbb{R}^{h \times w}$. Das liegt daran, dass pro Lage des Filters jeweils nur ein Pixel des neuen Bildes entsteht. Man kann sich vorstellen, dass an der Position des Quellenpixels jeweils ein neuer Pixel generiert wird. Jedoch muss der Filter immer eine vollständige Region als Rezeptives Feld besitzen. Dadurch kommt das Zentralelement nie auf die Ränder des Bildes zu liegen, wodurch sie entfallen.

Dieses ganze Verfahren der Filteroperation ist anschaulich in Abbildung (3.6) dargestellt.

Quellen: (12) (18)

3.3.4 Filteroperationen als diskrete Faltungen

Eigentlich handelt es sich bei der Anwendung eines Filters auf ein Bild um eine spezifische mathematische Operation: eine **diskrete Faltung**² (engl.: convolution).

Daher röhrt auch der Name des Convolutional Neural Networks. Da die Faltung als mathematische Operation relativ kompliziert ist, wird sie im Rahmen dieser Arbeit nicht in vollem Umfang behandelt. Ihre Bedeutung soll auf die Faltung von Filtern beschränkt werden.

Mithilfe der Faltungsoperation können die Schritte aus Sektion (3.3.3) zusammengefasst werden. Bei der Filteroperation handelt es sich nämlich um eine diskrete Faltung des Filtertensors \mathbf{W} über den Bildtensor \mathbf{B} . Somit kann folgende Formel verwendet werden, um einen Pixel $\tilde{\mathbf{B}}_{y,x}$ des neuen Bildes zu berechnen.

$$(\tilde{\mathbf{B}})_{y,x} = (\mathbf{W} * \mathbf{B})_{y,x} = \sum_{v=1}^f \sum_{u=1}^f (\mathbf{W})_{v,u} (\mathbf{B})_{y+v-1, x+u-1} \quad (3.2)$$

²Streng genommen wird in CNNs nicht eine diskrete Faltung durchgeführt, sondern eine sogenannte **Kreuzkorrelation**. Der Unterschied besteht darin, dass bei einer Kreuzkorrelation die Faltungsmatrix nicht horizontal und vertikal gespiegelt wird. Dies, im Gegensatz zu der eigentlichen Faltung, bei welcher diese Spiegelung stattfindet. Jedoch wird begrifflich das Wort Faltung für beide Operationen verwendet. Im Rahmen dieser Arbeit ist mit dem Wort Faltung immer die Kreuzkorrelation gemeint.

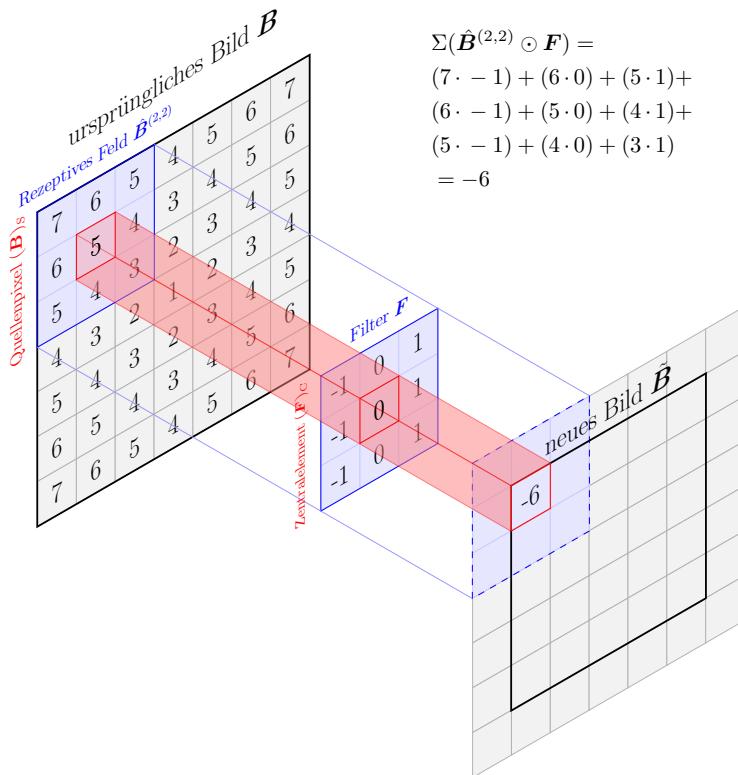


Abbildung 3.6: Schema, wie ein Filter über ein Bild läuft

Faltung Beispielrechnung

Es folgt nun eine Beispielrechnung für eine Faltungsoperation zweier Matrizen. Diese Matrizen haben folgende Einträge:

$$\mathbf{B} = \begin{pmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{pmatrix} \text{ und } \mathbf{W} = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

Um den ersten Pixel $(\tilde{\mathbf{B}})_{1,1}$ des Resultats zu erhalten, führen wir folgende Rechnung durch:

$$\begin{aligned}
 (\tilde{\mathbf{B}})_{1,1} &= (\mathbf{W} * \mathbf{B})_{1,1} = \sum_{v=1}^f \sum_{u=1}^f (\mathbf{W})_{u,v} (\mathbf{B})_{u,v} \\
 &= (10 \cdot 1) + (10 \cdot 0) + (10 \cdot -1) + (10 \cdot 1) + (10 \cdot 0) + (10 \cdot -1) + (10 \cdot 1) + (10 \cdot 0) + (10 \cdot -1) \\
 &= 0
 \end{aligned}$$

Das Gesamtergebnis lautet dann:

$$\tilde{\mathbf{B}} = \mathbf{W} * \mathbf{B} = \begin{pmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{pmatrix}$$

Für farbige Bilder ist das Vorgehen ähnlich. Es werden nun Tensoren dritten Ranges anstatt von Tensoren zweiten Ranges verwendet. Betrachten wird der allgemeine Bildtensor $\mathbf{B} \in \mathbb{R}^{h \times w \times c}$, wobei c die Anzahl Farbkomponenten (engl.: channels) ist. Nun benutzt man einen Filter $\mathbf{W} \in \mathbb{R}^{f \times f \times c}$ mit beliebiger Grösse f , aber

mit der gleichen Tiefe c , wie das Ursprungsbild \mathbf{B} . Dadurch muss der Filter nicht entlang der Tiefe des Bildes wandern, weil er die gleiche Tiefe aufweist. Das hat zur Folge, dass das verarbeitete Bild $\tilde{\mathbf{B}}$ immer eine Matrix ist. Die allgemeine Filtergleichung lautet:

$$(\tilde{\mathbf{B}})_{y,x} = (\mathbf{W} * \mathbf{B})_{y,x} = \sum_{v=1}^f \sum_{u=1}^f \sum_{w=1}^c (\mathbf{W})_{v,u,w} (\mathbf{B})_{x+u-1,y+v-1,w} \quad (\text{FO})$$

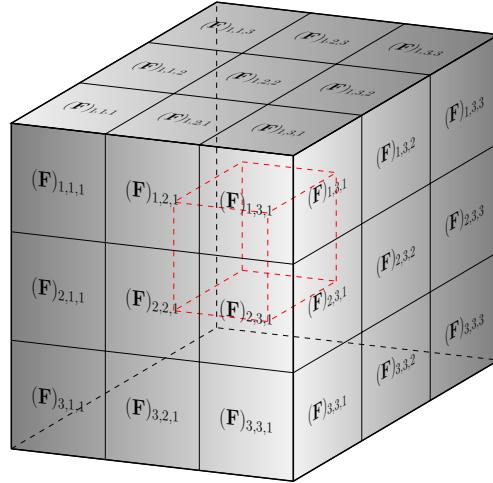


Abbildung 3.7: ein 3D-Filtertensor mit rotem Zentralelement $(\mathbf{F})_C$

Quellen: (1) (12) (13)

3.3.5 Mehrere Filter

Eine Filteroperation ist nicht auf einen einzigen Filter beschränkt. Es können mehrere Filter auf das gleiche Ausgangsbild angewendet werden und zusammen ein Endbild erzeugen.

Die Anzahl Filter werden mit c bezeichnet. Nun wird pro Filter \mathbf{W}_i eine Faltung über das Ursprungsbild \mathbf{W} gemacht, wobei jede Faltung eine neue Matrix $\tilde{\mathbf{B}}_i = \mathbf{W}_i * \mathbf{B}$ liefert. Dabei ist i der Index des Filters. All diese gefalteten Bilder $\tilde{\mathbf{B}}_i$ sind zwei-dimensionale Matrizen, unabhängig davon, wie viele Komponenten das Ursprungsbild hatte. Aus diesem Grund können die einzelnen Matrizen $\tilde{\mathbf{B}}_i$ aufeinander gelegt werden und somit einen grossen 3D-Endtensor $\tilde{\mathbf{B}}$ bilden. Hierfür müssen alle Filter \mathbf{W}_i gleich gross sein. Die Tiefe des neuen Bildes $\tilde{\mathbf{B}}$ ist jetzt gerade die Anzahl Filter c . Schon vorher wurde die Tiefe des Bildes (bzw. die Anzahl Farbkomponenten) mit c bezeichnet. Somit steht c sowohl die Anzahl Filter einer Schicht, wie auch für die Tiefe des verarbeiteten Bildes.

Quellen: (1) (12)

3.3.6 Padding

Wie bereits erwähnt schrumpfen die Bilder (an den Rändern), wenn man einen Filter auf sie anwendet. Zur Erinnerung: Dies liegt daran, dass pro Lage des Zentralelements jeweils nur ein Pixel des neuen Bildes entsteht. Das Zentralelement kommt jedoch nicht auf allen Ursprungspixeln zu liegen. Auf diese Weise fallen die Ränder weg. Der Umfang des Wegfalls hängt von der Filtergrösse f ab. Folgende Formel gilt für die neue Grösse n_1 des Bildes, berechnet anhand der alten Grösse n_0 . n bezeichnet eine der zwei Seitenlängen.

$$n_1 = n_0 - f + 1 \quad (3.3)$$

Das Problem hierbei ist, dass nach einigen Faltungen das Bild extrem geschrumpft ist und im Grenzfall kleiner als der Filter wird. Das darf natürlich nicht passieren. Zusätzlich kommt es zu einem relativen Informationsverlust an den Rändern, im Vergleich zum Rest des Bildes. Dies röhrt daher, dass es im Innern des Bildes zu mehr Überlappungen der Filterlagen kommt und dies bei den Rändern seltener auftritt.

Diese unerwünschten Phänomene lassen sich mit sogenanntem **Padding** beheben. Padding ist ein Vorgang, welcher vor der eigentlichen Faltung stattfindet. Dabei werden zusätzliche Ränder (Zeilen und Spalten) an das

0	0	0	0	0	0	0
0	5	4	3	4	5	0
0	4	3	2	3	4	0
0	3	2	1	2	3	0
0	4	3	2	3	4	0
0	5	4	3	4	5	0
0	0	0	0	0	0	0

Abbildung 3.8: Padding $p = 1$ in blau

Ursprungsbild angebracht. Der Tensor wird der Länge und der Breite (und nicht der Tiefe) entlang an den Enden erweitert. Die neuen Elemente werden dabei auf den Wert 0 gesetzt.

Das Padding p ist eine Zahl, welche angibt, wie viele Elemente an allen Rändern hinzugefügt werden. Padding $p = 1$ bedeutet, dass an allen Kanten jeweils eine Reihe bzw. Spalte hinzugefügt wird. Begrifflich unterscheidet man zwischen zwei Arten von Padding:

- **Valid-Padding:** Es werden keine zusätzlichen Elemente angebracht. p ist also 0.
- **Same-Padding:** Es werden so viele Reihen und Spalten angebracht, dass die Grösse des Bildes nach der Faltung unverändert bleibt.

Um Same-Padding durchzuführen, muss p so gewählt werden, dass es den Wegfall durch die Filteroperation gerade kompensiert. Da die Bilder durch das Padding der Länge und Breite nach jeweils um zwei p verlängert wird, erhält man den Ausdruck $n_1 = n_0 - f + 1 + 2p$. Wenn diese Formel nach p auflöst, erhält man folgende Formel für das Wählen des Same-Paddings p .

$$p = \frac{f - 1}{2} \quad (3.4)$$

Quellen: (12)

3.3.7 Stride

Bis jetzt wurde bei den Filterfaltungen der Filter pro Verschiebung immer nur um einen Pixel bewegt. Dies ist nicht zwingend. Grundsätzlich können Filter auch mit anderen Schrittgrössen verschoben werden. Diese Schrittgröße bezeichnet man als **Stride** s . Falls $s = 2$ gewählt wird, bedeutet das, dass der Filter sich pro Hadamard-Produkt um zwei Elemente verschiebt. Somit wurde eine Position übersprungen. Dies hat zur Folge, dass das neue Bild deutlich kleiner wird, denn das Zentralelement überspringt somit auch diese Felder und bildet so deutlich weniger Pixel.

7	6	5	4	5	6	7
6	5	4	3	4	5	6
5	4	3	2	3	4	5
4	3	2	1	2	3	4
5	4	3	2	3	4	5
6	5	4	3	4	5	6
7	6	5	4	5	6	7

Abbildung 3.9: Abbildung zum Stride: Ausgangsposition in blau, nächste Position mit $s = 1$ in rot, nächste Position mit $s = 2$ in grün

Folgende Formel beschreibt die Dimensionen des neuen Bildes unter Berücksichtigung der Filtergröße f , dem Padding p und dem Stride s .

$$n_1 = \frac{n_0 + 2p - f}{s} + 1 \quad (3.5)$$

Quellen: (12)

3.3.8 Vorzüge von Filtern

Es stellt sich nun die Frage, weshalb sich Filter für Maschinelles Lernen mit Bildern besonders eignen. Wie bereits erwähnt besteht die Aufgabe der Filter darin, bestimmte Features eines Bildes hervorzuheben und die restlichen auszublenden. Die gleiche Aufgabe erfüllen die Neuronen in einem KNN. Auch sie sollen Features der Inputdaten erlernen, wobei gewisse Neuronen auf gewisse Features reagieren. Weshalb verwendet man also nicht einfach KNNs (abgesehen von dem Problem mit der grossen Menge an Modellparametern, siehe Sektion (3))?

Man muss erkennen, dass sich Bilddaten deutlich von sonstigen Daten unterscheiden. Bildfeatures bzw. Pixel sind nur im Kontext ihrer Nachbarn relevant. Denn ein Pixel ist erst dann eine Kante oder eine Ecke, wenn er zwei verschiedene Farbregionen voneinander trennt. Oder ein Gegenstand wird erst durch eine ganze Anzahl von Pixeln und deren relative Position zueinander charakterisiert. Man bezeichnet diesen Umstand als **lokalierte Features**.

Des Weiteren sind Bildausschnitte nicht immer gleich ausgerichtet. Wenn man ein Gesicht auf einem Bild erkennen möchte, sollte es aber keine Rolle spielen, wo es sich auf dem Bild befindet, welche Grösse es hat und welche Ausrichtung es aufweist. Um diese Eigenschaften irrelevant für das Modell zu machen, muss es gewisse **Invarianzen** erfüllen.

Der Wesenszug, welche CNNs für lokale Features und Invarianzen geeignet macht, bezeichnet man als **Parameter-Sharing**. Er bezeichnet den Umstand, dass auf mehrere oder alle Features die gleichen Modellparameter wirken. Bei CNNs wird dies durch die Bewegung des Filters bzw. seiner Einträge über (fast) alle Pixel bewerkstelligt. Somit ist es egal, wo und wie sich die lokalen Features befinden. Dies führt dann zu den gewünschten Invarianzen: Translations-, Rotations- und Helligkeitsinvarianz.

Ein weiterer Vorteil des Parameter-Sharing ist, dass die Inputbilder beliebige Dimensionen besitzen können, da die Filter ihre Bewegung lediglich an die Grösse des Bildes anpassen müssen.

Quellen: (12)

3.3.9 Convolutional-Schicht

Nun soll zusammengeführt werden, was soeben über Filter erläutert wurde, um die Convolutional-Schicht zu definieren.

Die Convolutional-Schicht l beginnt mit ihrem Input, also den Aktivierungen $\mathbf{A}^{l-1} \in \mathbb{R}^{h^l \times w^l \times c^l}$, welche die vorherige Schicht ($l-1$) produziert hat. Falls es sich um die erste Schicht handelt, erhält sie den Input \mathbf{X} des Netzes.

Die Schicht besitzt c^l Varianten an Filtern $\mathbf{W}_i^l \in \mathbb{R}^{f^l \times f^l \times c^{l-1}}$, wobei i der Index ist. Diese Filter haben alle die gleichen Eigenschaften bezüglich: der Grösse f^l , der Tiefe c^{l-1} , dem Padding p^l und dem Stride s^l . Sie unterscheiden sich nur in den Modellparametern.

$$\mathbf{W}_i^l = \left[\begin{pmatrix} w_{i|1,1,1}^l & \cdots & w_{i|1,f,1}^l \\ \vdots & \ddots & \vdots \\ w_{i|f,1,1}^l & \cdots & w_{i|f,f,1}^l \end{pmatrix} \quad \cdots \quad \begin{pmatrix} w_{i|1,1,c^l}^l & \cdots & w_{i|1,f,c^l}^l \\ \vdots & \ddots & \vdots \\ w_{i|f,1,c^l}^l & \cdots & w_{i|f,f,c^l}^l \end{pmatrix} \right]$$

Nun wird jeder Filter \mathbf{W}_i^l einzeln über das Bild \mathbf{A}^l gefaltet, wodurch mehrere neue 2D-Bilder $\tilde{\mathbf{A}}_i^l$ entstehen.

$$\tilde{\mathbf{A}}_i^l = \mathbf{W}_i^l * \mathbf{A}^l \tag{3.6}$$

Die Faltung berechnet sich aus folgender Gleichung:

$$(\tilde{\mathbf{A}}_i)^l_{y,x} = (\mathbf{W}_i^l * \mathbf{A}^l)_{y,x} = \sum_{v=1}^f \sum_{u=1}^f \sum_{w=1}^c (\mathbf{W}_i^l)_{v,u,w} (\mathbf{A}^l)_{x+v-1,y+u-1,w} \tag{FO}$$

Jede dieser Matrizen ist ein Querschnitt $\tilde{\mathbf{A}}_{\cdot,\cdot,i}^l$ entlang der Tiefe des neuen 3D-Tensors $\tilde{\mathbf{A}}^l \in \mathbb{R}^{h^{l+1} \times w^{l+1} \times c^l}$. Hierbei ist die Tiefe c^l gerade die Anzahl der Filter c^l . Die Höhe h^{l+1} und die Breite w^{l+1} werden jeweils gemäss nachstehender Formel berechnet, anhand der Höhe h^l und der Breite w^l des Ausgangsbildes:

$$n^{l+1} = \frac{n^l + 2p^l - f^l}{s^l} + 1 \tag{3.7}$$

Da die Faltungsoperation linear ist, wird wiederum eine nicht-lineare Aktivierungsfunktion benötigt, um das

Modell zu befähigen, nicht-lineare Probleme zu lösen. Deshalb wird in einem letzten Schritt die vektorisierte Aktivierungsfunktion φ auf den Tensor $\tilde{\mathbf{A}}^l$ angewendet. So ergibt sich die neue Aktivierung \mathbf{A}^{l+1} der nächsten Schicht. Empirisch lässt sich nachweisen, dass sich für CNNs die ReLU-Aktivierungsfunktion aus Sektion (2.2.3) besser eignet als die Sigmoidfunktion. Deshalb wird in der Regel die ReLU-Funktion für CNNs verwendet.

$$\mathbf{A}^{l+1} = \varphi[\tilde{\mathbf{A}}^l] \quad (3.8)$$

Quellen: (13) (12) (1) (3)

3.4 Dimensionalitätskontrolle

Beim Anwenden einer Convolutional Schicht gehen Informationen verloren, da nur die relevanten Features hervorgehoben werden und der Rest verworfen wird. Jedoch schrumpft das Bild entweder gar nicht (bei Same-Padding) oder es schrumpft nur vergleichsweise leicht (bei Valid-Padding). Dies ist ein Problem, da die Information in deutlich weniger Pixeln bzw. Tensorelementen codiert werden könnte. In der Konsequenz zeigt sich ein unnötig hoher Ressourcenverbrauch und eine erhöhte Gefahr für Overfitting (siehe Sektion (1.3.2)). Um diesem Effekt entgegenzuwirken, gibt es einerseits sogenannte **Pooling-Schichten** und als Gegenstück dazu **Upsampling-Schichten**. Ersteres wird verwendet, um die Dimensionalität der Bilder zu vermindern und Letzteres, um die Dimensionalität der Bilder zu erweitern. Somit ermöglichen diese Schichten eine kontrollierte Art, (im Gegensatz zum Padding) die Dimensionalität willentlich zu bestimmen. Diese Fähigkeit ist essenziell, um in Sektion (4.4) die Topologie eines Convolutional Autoencoders zu realisieren.

3.4.1 Pooling-Schicht

Die Pooling-Schicht verringert die Dimensionalität durch das Zusammenfassen eines Feldes von Tensorelementen zu einem einzigen Tensorelement. Dabei wandert das Feld nur entlang der Länge und Breite des Bildes und nicht etwa entlang der Tiefe. Dies bedeutet, dass das Pooling auf jede Farbkomponente einzeln angewendet wird, wodurch sich die Tiefe des Bildes nicht ändert.

Für die Beschreibung einer Pooling-Schicht finden die gleichen Begriffe Anwendung wie bei der Convolutional-Schicht und den Filtern. Die Grösse des Elementefeldes, welches zusammengefasst wird, bezeichnet man analog zur Filtergrösse mit f^l . Das Stride s^l bezeichnet auch beim Pooling, wie gross die Schrittgrösse beim Verschieben des Feldes ist. Meistens wählt man den Stride s^l gerade gleich der Feldgrösse f^l , damit alle Pixel zusammengefasst werden. Kleiner als f^l kann s^l nicht gewählt werden.

Man unterscheidet zwischen zwei Arten von Pooling:

- **Average-Pooling:** Das Elementefeld wird zusammengefasst, indem das arithmetische Mittel der Elemente gebildet wird.
- **Max-Pooling:** Das Elementefeld wird zusammengefasst, indem das Element mit dem höchsten Wert beibehalten wird und die anderen verworfen werden.

In der Praxis verwendet man eigentlich nur Max-Pooling, da es deutlich bessere Resultate erzielt.

MaxPooling Beispiel

Es sei eine MaxPooling-Schicht gewählt mit $f^l = 2$ und $s^l = 2$. Diese fasst also jedes (2×2) -Feld zu einem Element zusammen. Da aus vier Elementen jeweils eines wird, entspricht dies einer Informationsreduktion von 75%.

$$\begin{pmatrix} 5 & 2 & 4 & 3 \\ 8 & 9 & 5 & 1 \\ 3 & 8 & 6 & 7 \\ 8 & 1 & 4 & 2 \end{pmatrix} \xrightarrow{\text{MaxPool}} \begin{pmatrix} 9 & 5 \\ 8 & 7 \end{pmatrix}$$

Es ist festzuhalten, dass eine Pooling-Schicht keinerlei Modellparameter besitzt. Somit gibt es nichts zu trainieren. Sie besitzt lediglich einige Hyperparameter, wie die Feldgrösse f^l und den Stride s^l .

Um zu berechnen, was die neuen Dimensionen des Bildes nach dem Pooling sind, gelten die gleichen Formeln wie für die Filteroperationen:

$$n_1 = \frac{n_0 - f}{s} + 1 \quad (3.9)$$

Quellen: (12) (1)

3.4.2 Upsampling-Schicht

Die Upsampling-Schicht bildet das Gegenstück zur Pooling-Schicht, da sie die Dimensionalität des Bildes erhöht. Auf den ersten Blick erscheint unklar, weshalb man die Dimensionalität erhöhen will, da die eigentliche Motivation dieser Schichten in der Verhinderung von Overfitting besteht. Jedoch sollte mit dem Kapitel ④ klar werden, weshalb solche Schichten höheren Nutzen haben können.

Man verwendet auch hier wieder den Begriff der Feldgrösse f^l . Dieses Mal beschreibt der Wert, wie stark das Bild hochskaliert wird. Beim Upsampling wird ein Element zu einem $(f^l \times f^l)$ -Feld hochgerechnet und erweitert so die Dimensionalität.

Es stellt sich die Frage, welche Werte das neue Feld erhält. Dafür ist eine Interpolationsart zu wählen. Zwei Arten sind besonders verbreitet:

- **Bilineare Interpolation**
- **Nächste-Nachbar-Interpolation** (engl.: nearest-neighbor-interpolation)

Hier soll nur die Nächste-Nachbar-Interpolation betrachten werden. Sie ist vergleichsweise trivial, denn alle neuen Feldelemente nehmen einfach den Wert des alten Elements an. Der Wert wird auf diese Weise vermehrt.

Quellen: (12) (1)

Kapitel 4

Autoencoder

In Kapitel drei wurde die Architektur eines Convolutional Neural Networks dargelegt. Nun soll eine weitere Architektur erörtert werden, welche das letzte Bindeglied zur praktischen Programmierung eines Convolutional-Denoising-Autoencoder verkörpert. Dieses ist notwendig, um die theoretische Funktionsweise eines Autoencoders zu verstehen. Zielsetzung ist, den Autoencoder mit dem Convolutional Neural Network zu fusionieren, um auf diese Weise einen Convolutional-Autoencoder zu erhalten. Zu guter Letzt soll dann noch eine Anwendung eines solchen Convolutional-Autoencoder betrachtet werden, in der Form eines Denoiser.

Ein **Autoencoder** ist eine Architektur, welche ein KNN nicht bezüglich der Schichtenarten, sondern auf der Ebene der Netzform beschreibt. Die Aufgabe eines Autoencoders besteht darin, eine neue Repräsentation einer Datenmenge zu erlernen. Diese neuartige Darstellung soll mit weniger Daten möglichst die gleiche Information codieren. Somit entwickelt das Modell eine **effiziente Daten-Codierung** für einen Datensatz. Dadurch kann er zur **Dimensionalitätsreduktion** verwendet werden. Neben dieser neuen Repräsentation erlernt das Netzwerk darüber hinaus, wie es aus dieser Codierung wieder die Ursprungsdaten reproduzieren kann.

Autoencoder gehören nicht dem klassischen überwachtem Lernen an. Streng genommen zählen sie zum unüberwachten Lernen, da in den Trainingsdaten keine Labels enthalten sein müssen. Jedoch gilt das gleiche Vorgehen, wie bei Überwachtem Lernen. Das Netzwerk wird nämlich ebenfalls darauf trainiert, gewisse gewünschte Outputs zu liefern. Nur sind in diesem Fall die “Labels” gleich den Features!

Quellen: (19)

4.1 Topologie

Ein Autoencoder besteht ebenso wie das klassische KNN aus Neuronen. Wie bereits erwähnt sind die “Labels” eines Autoencoders gleich den Features. Somit versucht ein Autoencoder, die Werte der Inputneuronen möglichst exakt in die Outputneuronen zu kopieren. Diese Operation wäre an sich ziemlich bedeutungslos, da mit den Daten nichts passiert. Das Netzwerk würde lediglich erlernen, eine Identitätsfunktion zu imitieren. Aus diesem Grund muss man gewisse Einschränkungen einführen. Diese bringen das Netzwerk dazu, interessante Methoden zu entwickeln, um die Features *approximativ* zu rekonstruieren.

Die angesprochene Einschränkung besteht darin, dass dem Autoencoder nicht beliebig viel Kapazität für die Codierung der Features gewährt wird. Es stehen dem Netzwerk nur wenige Zahlenwerte zur Verfügung, um die Features in der neuen Repräsentation zwischenzuspeichern, bevor sie wieder in die ursprüngliche Form zurück transformiert werden. Die Kapazitätseinschränkung wird durch die Topologie des Autoencoders bezweckt.

Der einfachste Autoencoder besteht aus drei Schichten, welche sich wie folgt gliedern lassen:

- eine Inputschicht mit d Neuronen
- eine Zwischenschicht mit p Neuronen, bezeichnet als **Flaschenhals**
- eine Outputschicht mit d Neuronen

In Abbildung (4.1) ist ein beispielhafter Autoencoder abgebildet, mit $d = 5$ Features und einem Flaschenhals der Grösse $p = 2$.

Die Inputschicht enthält die Features $\mathbf{x} \in \mathbb{R}^d$. Dagegen enthält die Outputschicht die **Rekonstruktionen** der Features, welche wir mit $\hat{\mathbf{x}} \in \mathbb{R}^d$ bezeichnen. Als logische Konsequenz müssen die beiden Schichten die gleiche

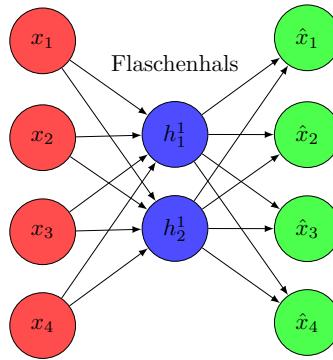


Abbildung 4.1: Schichten eines kurzen Autoencoders

Anzahl d an Neuronen besitzen. Die Flaschenhalsschicht hingegen besitzt nur p Neuronen, wobei $p \ll d$ ist. Somit ist sie deutlich kleiner als die anderen beiden Schichten und bildet damit die **Kapazitätsbeschränkung**.

Der soeben dargestellte Autoencoder besitzt nur eine Zwischenschicht. Gängiger ist jedoch, das Netz aus mehreren Zwischenschichten zu konstruieren. Dabei werden die Schichten zum Flaschenhals hin immer schmäler und die Schichten nach dem Flaschenhals wieder dicker. In Abbildung 4.2 ist dies gut ersichtlich. Aus Vereinfachungsgründen soll dennoch die Funktionsweise eines Autoencoders mit nur einer Zwischenschicht hergeleitet werden.

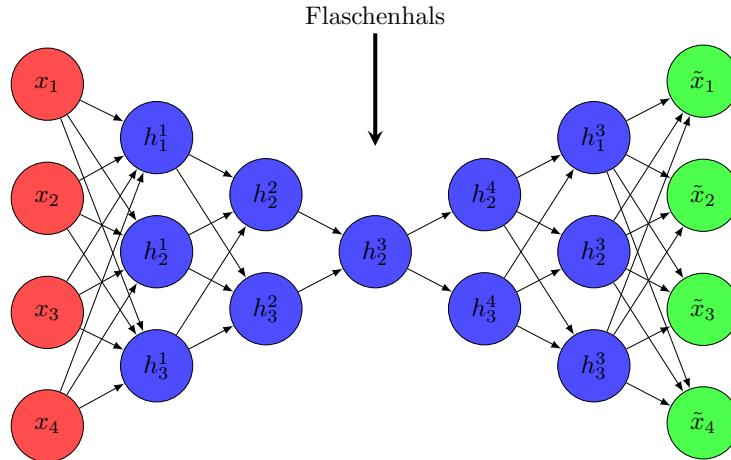


Abbildung 4.2: Schichten eines tieferen Autoencoders

4.2 Funktionsweise

Ein Autoencoder möchte die Features \mathbf{x} von der Inputschicht in die Outputschicht kopieren. Jedoch zwingt der Flaschenhals den Autoencoder dazu, eine neue Repräsentation der Features zu erlernen. Dies aus dem Grund, weil die Features in ihrer jetzigen Form keinen Platz im Flaschenhals finden, da $p \ll d$. Daher muss das Modell die Informationen der Features auf nur einige wenige relevante Eigenschaften komprimieren. Das Modell muss gewissermassen einen Code entwickeln. Diese neue Repräsentation nennt man das **Encoding**.

In einem zweiten Schritt muss der Autoencoder aus diesem Encoding wiederum versuchen, die ursprünglichen Features zu rekonstruieren. Diese Rekonstruktion bezeichnet man als das **Decoding**.

Encoder und Decoder

Ein Autoencoder in zwei TeilmODELLE untergliedern werden. Das eine Modell erzeugt das Encoding und das andere das Decoding. Dafür wird die Hypothesenfunktion h in ein Funktionenpaar (ϕ, ψ) aufgespalten.

- $\phi : \mathcal{X} \rightarrow \mathcal{E}$ Encoder-Funktion
- $\psi : \mathcal{E} \rightarrow \mathcal{X}$ Decoder-Funktion

Die erste Teilfunktion ϕ ist für das Encoding zuständig. Sie bildet den Inputraum $\mathcal{X} \subseteq \mathbb{R}^d$ auf den Encodingraum $\mathcal{E} \subseteq \mathbb{R}^p$ ab, welcher die neue Repräsentation \mathbf{x}^* enthält. Der Inputraum \mathcal{X} enthält sämtliche Features \mathbf{x} .

Das Gegenstück ist die Teilefunktion ψ . Sie bildet den Encodingraum \mathcal{E} zurück auf den Inputraum \mathcal{X} ab.

Der Output des Autoencoders $\hat{\mathbf{x}}$ wird somit durch die aufeinander folgende Anwendung der Teilefunktionen auf die Features \mathbf{x} gebildet:

$$\hat{\mathbf{x}} = \psi(\phi(\mathbf{x})) = (\psi \circ \phi)(\mathbf{x}) \quad (4.1)$$

Beim Trainieren wird die Kostenfunktion des Modells minimiert. Beispielsweise kann dafür die MSE-Kostenfunktion C_{MSE} dienen. Wird sie als Kostenfunktion verwendet, so wird folgender Ausdruck minimiert:

$$\min_{\phi, \psi} (\mathbf{x} - \hat{\mathbf{x}})^2 = \min_{\phi, \psi} (\mathbf{x} - (\psi \circ \phi)(\mathbf{x}))^2 \quad (4.2)$$

Anders ausgedrückt wird versucht, Funktionen ϕ und ψ zu finden, welche die Differenz zwischen den ursprünglichen Features \mathbf{x} und den Rekonstruktionen $\hat{\mathbf{x}}$ minimieren. Dabei muss der Autoencoder eine Dimensionalitätsreduktion betreiben. Er entwickelt einen Code, um die Features in der Flaschenhalsschicht komprimiert zu repräsentieren. Durch die Kapazitätseinschränkung $p \ll d$ muss \mathcal{E} eine niedrig-dimensionale Codierung der Features darstellen. Dem Modell ist es nicht möglich, die Inputs identisch wiederherzustellen, da bei der Abbildung ϕ Informationen verloren gegangen sind. Dadurch sollte der Code im optimalen Fall nur noch die Merkmale der Features umfassen, welche die umfangreichsten Informationen beinhalten. Nur so kann die bestmögliche Rekonstruktion vollzogen werden, um minimale Kosten zu erzeugen. Dieser Code kann extrahiert werden, um eine Repräsentation der Features zu erhalten, welche nicht mehr mehrere hundert Merkmale umfasst, sondern nur noch einige wenige. Jegliche natürliche Schwankung in den Werten der Features geht verloren, die nicht verallgemeinert werden kann. Solche natürlichen Schwankungen bezeichnet man als **Datenrauschen**.

Der Code

Es ist wichtig zu erkennen, dass der Code, welcher der Autoencoder entwickelt, **datenspezifisch** ist. Das heißt, er kann nicht wie z.B. ein Bildkompressionsalgorithmus auf beliebige Daten angewendet werden. Stattdessen ist er nur sinnvoll für die Kompression von Daten verwendbar, mit welchen er trainiert wurde.

Die Codeentwicklung eines Autoencoders kann man sich ungefähr so vorstellen: Ein Autoencoder wird mit einem Datensatz von Bildern menschlicher Gesichter trainiert. Anstatt, dass der Autoencoder die Farbkomponenten der Pixel im Encoding speichert, würde er die Features abstrahieren. Er würde Information zum Augenabstand, zur Nasenbreite, zur Augenfarbe usw. als Code verwenden, welche einen deutlich geringeren Informationsgehalt aufweisen als sämtliche Pixel. Diese Informationen reichen aber aus, um relativ gute Rekonstruktionen zu erzeugen.

4.3 Anwendungen

Die Eigenschaft, welche einen Autoencoder für die diverse Anwendungen geeignet macht, ist die Dimensionalitätsreduktion. Wir werden nun eine Anwendung grob betrachten und eine weitere detailliert in der nächsten Sektion: der Denoiser.

4.3.1 Datenkompression

Durch die Dimensionalitätsreduktion besitzt das Encoding einen geringeren Informationsgehalt als die ursprünglichen Daten. Aus diesem Grund kann der Autoencoder zur Datenkompression verwendet werden. Die komprimierten Daten können somit platzsparender gespeichert oder über das Internet versendet werden. Die komprimierten Daten benötigen um den Faktor $\frac{p}{d}$ weniger Speicher, wobei p die Größe des Flaschenhalses und d die Größe der Inputschicht ist. Mithilfe des Decoders können die ursprünglichen Daten zurückgewonnen werden. Jedoch hat das Decoding einen gewissen Qualitätsverlust im Vergleich zu den Features erlitten. Somit können die Daten nicht in voller Qualität rekonstruiert werden. Es handelt sich also um eine **verlustbehaftete Kompression**. Außerdem ist diese Kompression datenspezifisch. Das heißt, dass ein Autoencoder, welcher darauf trainiert wurde, Bilder von Katzen zu komprimieren, nur sehr schlechte Rekonstruktionen für Landschaftsbilder erzeugen würde. Dies steht im Gegensatz zu einer JPEG-Kompression, welche sich für beliebige Bilder eignet, aber geringere Kompressionsraten aufweist als Autoencoder.

Eine andere datenspezifische Kompressionsmethode stellt die **Hauptkomponentenanalyse** (PCA) dar. Sie ist eine erweiterte Version der Hauptachsentransformation, welche benötigt wird, um Kegelschnitte in Standardlage zu bringen. Falls ausschließlich lineare Neuronen in einem Autoencoder verwendet werden, ist das Verhalten

des Modells sehr ähnlich zu PCA. Jedoch sind Autoencoder grundsätzlich PCAs überlegen, da sie mithilfe von nicht-linearen Neuronen komplexere Kompressionsmethoden entwickeln können.

Quellen: (20) (21)

4.4 Convolutional-Autoencoder

Ein Convolutional-Autoencoder ist, wie es der Name schon andeutet, ein Autoencoder, welcher anstatt der normalen Fully-Connected-Schichten eines KNNs, die Schichten eines CNN verwendet. Dieser ist interessant, weil er zur Dimensionalitätsreduktion von Bildern verwendet werden kann. Er kann beispielsweise als Bildkompressionsalgorithmus genutzt werden.

Wie bei den normalen Autoencodern auch, müssen hier die Inputschicht und das Decoding die gleiche Form aufweisen. Nun ist auch ersichtlich, weshalb es in Sektion (3.4) wichtig war, CNN-Schichten zu entwickeln, welche ein willentliches Einstellen der Schichtgrößen erlauben. Es muss versucht werden, dass die Vergrößerung nach dem Flaschenhals die Verkleinerung vor dem Flaschenhals kompensiert. Eine Massnahme dafür ist, bei sämtlichen Convolutional-Schichten Same-Padding zu verwenden und die Dimensionalität nur über Pooling- und UpSampling-Schichten zu steuern. Des Weiteren muss der Decoder den spiegelverkehrten Aufbau zum Encoder aufweisen, wobei Pooling-Schichten durch UpSampling-Schichten ersetzt werden.

In Abbildung (4.3) ist ein schematischer Convolutional Autoencoder aufgezeigt. Dieser verarbeitet ein farbiges Bild mit drei Farbkomponenten.

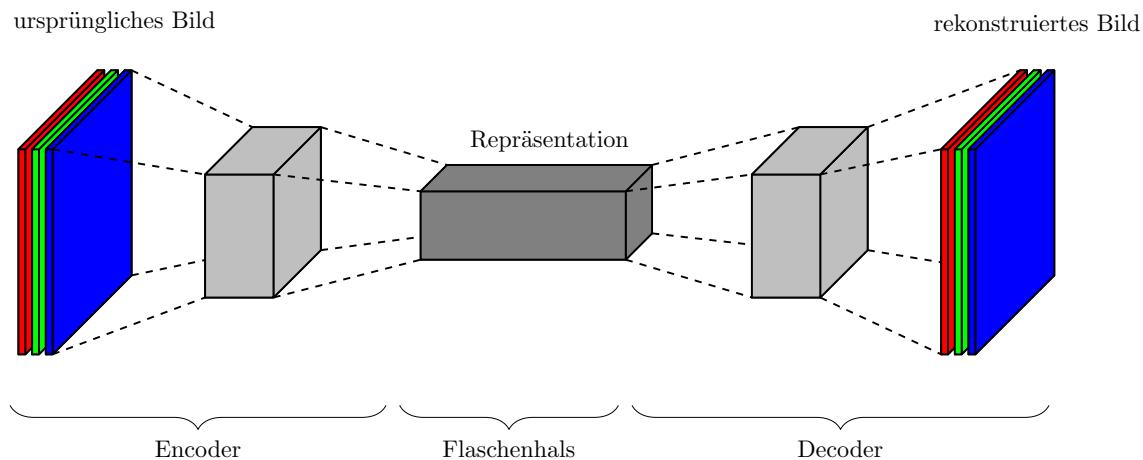


Abbildung 4.3: Darstellung eines Convolutional Autoencoders

Ausserdem sind Convolutional-Autoencoder von Interesse, da sie zu einer weiteren Anwendung führen: zum Denoising-Autoencoder. Dieser wird im nächsten Abschnitt betrachtet.

Quellen: (22)

4.5 Denoising-Autoencoder

Ein Denoising-Autoencoder oder einfach Denoiser verkörpert einen leicht abgeänderten Autoencoder. Er wird vor allem verwendet, um das Bildrauschen aus Bildern zu entfernen.

Im Unterschied zu klassischen Autoencodern benötigt er einen Datensatz mit wahren Labels, da er nicht die Features kopiert. Seine Inputs bestehen aus den verrauschten Bildern. Die Labels sind die unverrauschten Bilder. Somit wird er darauf trainiert, das Rauschen zu entfernen. Man mag sich nun die Frage stellen, weshalb hierfür ein Autoencoder und kein normales CNN verwendet wird. Empirisch hat sich gezeigt, dass Denoiser besser für das Entrauschen von Bildern geeignet sind. Das hat nachvollziehbare Gründe. Bei der Entwicklung des Codes für Encoding-Repräsentation, kann das Rauschen nicht codiert werden, da es völlig zufällig auftritt. Es existieren keine Muster oder Gesetzmäßigkeiten, welche anders codiert werden könnten. Somit muss der Autoencoder diese Rausch-Features verwerfen. Gegenüber sonstigen CNNs hat er ein leichteres Spiel.

In Abbildung (4.4) ist ein Denoiser abgebildet, welcher ein graustufiges Bild mit nur einer Farbkomponente entrauscht.

Quellen: (23)

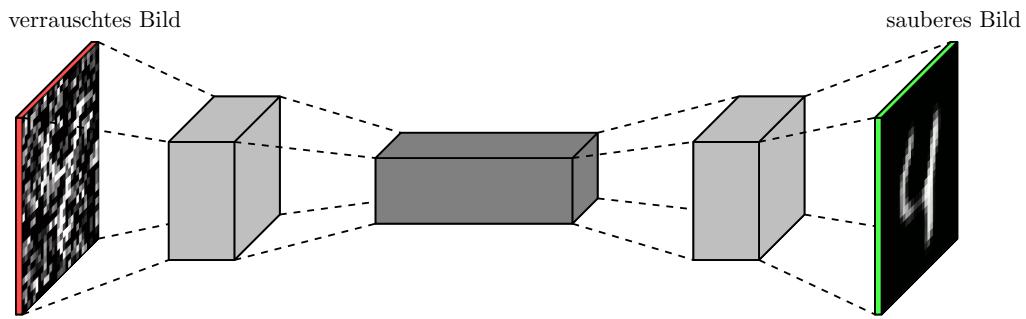


Abbildung 4.4: Visualisierung eines Convolutional-Denoising-Autoencoder

4.5.1 Generierung der verrauschten Bilder

Ein Denoising-Autoencoder kann verwendet werden, um diverse Arten von Rauschen zu entfernen. Jedoch werden wir uns in dieser Arbeit auf ein generiertes Rauschen beschränken: das additive Gauss'sche Rauschen. Um es zu erzeugen, wird eine Gauss'sche Normalverteilung $\mathcal{N}(\mu = 0, \sigma^2 = 1)$ verwendet mit Erwartungswert $\mu = 0$ und Varianz $\sigma^2 = 1$. Für jedes Feature des Datensatzes wird dieser Verteilung ein Zufallswert entnommen. Dieser Wert wird mit einem Rauschfaktor c multipliziert, welcher die Rauschstärke bestimmt. Das Resultat wird zum entsprechenden Wert des Features addiert. Auf diese Weise lässt sich ein verrauschter Datensatz entwickeln.

$$r \sim \mathcal{N}(\mu = 0, \sigma^2 = 1)$$

$$\tilde{x} = x + c \cdot r$$

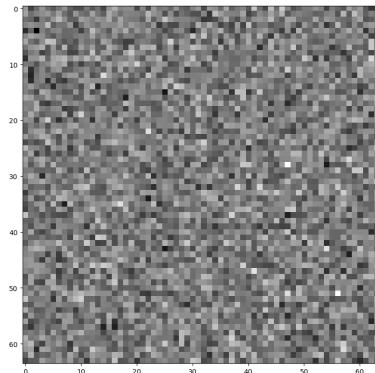


Abbildung 4.5: zwei-dimensionales Gauss'sches Rauschen mit Rauschfaktor $c = 1$

Quelle: (24)

Kapitel 5

Frameworks für Maschinelles Lernen

In den Kapiteln eins bis vier wurde die theoretische Funktionsweise von Maschinellem Lernen anhand von zwei konkreten Architekturen (CNN und Autoencoder) eines KNNs dargelegt. Nun findet ein Übergang zur praktischen Programmierung eines Convolutional-Denoising-Autoencoders statt. Es wird hier die grobe Funktionsweise der Frameworks TensorFlow und Keras erörtert. Aus didaktischen Gründen wird die Verwendung isoliert betrachtet. Sie dient als Vorbereitung auf das Kapitel sechs, wo der Gesamtzusammenhang anhand der fallbezogenen Programmierung hergestellt wird.

5.1 TensorFlow

TensorFlow (TF) ist ein von Google entwickeltes **Framework für datenstromorientierte Programmierung**. Es ist im Grunde genommen eine Mathematik-Programmbibliothek für numerische Berechnungen, dessen Hauptanwendungsbereich im Maschinellen Lernen liegt. TF zeichnet sich hauptsächlich durch seine einfache Bedienung aus, welche es dem Benutzer erlaubt, ohne grosse mathematische Vorkenntnisse, ein ML-Modell zu entwickeln. Des Weiteren sind die performanten Implementierungen bemerkenswert, welche den Trainingsvorgang sehr kurz halten. Daher ist TF gegenüber selbst geschriebenen Implementierungen zu bevorzugen. TF ist gratis und ein Open-Source-Projekt. Es steht auf Github zur Verfügung (siehe QR-Code). In der Industrie ist TensorFlow gut etabliert und weit verbreitet. Es wird von vielen grossen Internetunternehmen verwendet. Einige dieser Firmen sind: Google, Twitter, AirBnB, Intel, Snapchat und noch viele mehr.



Grundsätzlich wurde TF für Maschinelles Lernen entwickelt. Speziell eignet es sich für Deep Learning. Es bietet die nötigen Werkzeuge, um ein Modell zu definieren und es mit den eingebauten Algorithmen zu trainieren. Trotz des engen Bezugs zum Maschinellen Lernen, eignet sich TF darüber hinaus auch für andersartige Anwendungen, welche Gebrauch von numerischen Berechnungen machen.

Abbildung 5.1: Im Folgenden wird die grobe Funktionsweise von TensorFlow untersucht. Wir beziehen uns dabei auf TensorFlow-Core Version r1.14¹. Die Betrachtung von TF wird hier bewusst kurz gehalten, da bei der Programmierung in Kapitel sechs auf die High-Level-API Keras zurückgegriffen wird. Mit TF wird daher nur teilweise direkt gearbeitet. Trotzdem ist es hilfreich, ein großes Verständnis über TF zu besitzen, um die Funktionsweise zu verstehen.

QR-Code zum GitHub-Repo

Die wichtigste Quelle, um TensorFlow zu erlernen, ist die offizielle TensorFlow Website. Dort findet man auch die Dokumentation.

Quellen: (25) (26)

5.1.1 Frontend und Backend

Wie die meisten grossen Frameworks ist TensorFlow in Backend und Frontend aufgeteilt.

¹Momentan befindet sich TF 2.0 in der Beta Version. Es handelt sich dabei um eine grosse Überarbeitung des Frameworks. Sobald TF 2.0 offiziell veröffentlicht wurde, ist es zu empfehlen, diese zu verwenden.

Fontend und Backend

Das **Frontend** ist der Teil eines Programms, mit welchem der Benutzer interagiert. Es stellt eine Schnittstelle zum sogenannten Backend eines Programms dar. Es offenbart so gewisse Funktionalitäten des Programms, ohne die zugrunde liegenden Logik preiszugeben.

Das **Backend** ist das Gegenstück zum Frontend. Es implementiert jegliche Programmlogik, welche das gewünschte Verhalten der Applikation hervorruft. Der Benutzer hat keinen direkten Zugriff auf diesen Teil des Programms und bekommt somit nichts direkt von dessen Existenz mit.



Das Frontend von TF ist in **Python** geschrieben². Python wird allgemein als eine sehr einsteigerfreundliche Programmiersprache wahrgenommen. Sie ermöglicht es dem Benutzer, mit vergleichsweise wenig Programmcode TensorFlow intuitiv zu verwenden. In Kapitel sechs wird die Entwicklung eines Modells in Python vorgenommen.

Das Backend von TF ist dagegen in **C++** (und CUDA) geschrieben. C++ ist im Gegensatz zu Python sehr performant. Daher ist die Ausführungszeit eines Programms sehr kurz. In der Abbildung 5.2: Konsequenz ist der Programmcode deutlich anspruchsvoller und ausführlicher (engl.: verbose) geschrieben. Da der Benutzer jedoch keine direkte Interaktion mit dem Backend hat, muss er auch nicht mit C++ vertraut sein.

Die hohe Performance von TF wird durch verschiedene Designentscheidungen garantiert. Falls der Leser am besseren Verständnis dieser Entscheide und der eigentlichen Implementierung des Backends interessiert ist, findet er im Anhang C weitere Informationen dazu vor.

5.1.2 Graph

TensorFlow ist datenstromorientiert, was bedeutet, dass eine Berechnung durch einen gerichteten Graphen beschrieben wird. Zuerst wird ein solcher Graph aufgestellt und es werden Knoten und Pfade hinzugefügt. Erst nach dieser Definition wird er ausgeführt. Es handelt sich dabei um einen **Computational Graph**, welcher schon in Anhang B erwähnt wurde. In TensorFlow wird er vereinfacht als **Graph** bezeichnet.

Dieser Graph führt eine Datenstrom-Berechnung aus. Er besteht aus einer Menge an Knoten, welche miteinander über Pfade verbunden sind. Die Knoten und die Pfade bilden die Operationen, welche auf die Zahlenwerte angewandt werden. Die Resultate landen jeweils im neuen Knoten. Jeder Knoten hat null oder mehr Inputs und null oder mehr Outputs. Zur Definition des Graphs müssen lediglich die Operationen festgelegt werden.

Mithilfe des Graphs wird das eigentliche ML-Modell definiert. Die Ausführung von ihm stellt die Vorwärtspropagation dar.

5.1.2.1 Knoten als Tensoren

Die zentrale Dateneinheit in TF sind Tensoren, welche bereits in Sektion 3.1 dargelegt wurden. Auch die Knoten des Graphen verkörpern Tensoren. Die meisten jedoch enthalten keine festen Zahlenwerte, sondern werden erst bei der Ausführung des Graphen mit Werten gespeist. Danach können die Werte ausgelesen werden.

Die Tensoren erlauben es den Daten, innerhalb des Graphen beliebige Dimensionen und Formen anzunehmen. Es können sowohl Listen von Zahlen (Vektoren), wie auch Bilder (Tensoren dritter Stufe) verrechnet werden. Aus diesem Grund war es auch sinnvoll, in Sektion 2.5 und im Anhang B die Gleichungen in Matrixform herzuleiten.

Ein **Tensor** ist definiert durch seine Form und den Datentyp seiner Elemente. Fast alle Tensoren sind mit einer einzigen Graph-Ausführung assoziiert. Sie besitzen ihren Wert nur für den jeweiligen Durchlauf, danach wird er verworfen. Man sagt die Tensoren sind “immutable”, da die Werte nicht erhalten bleiben und daher auch nicht modifiziert werden können.

Die meisten Knoten innerhalb des Graphen verkörpern Operationen, jedoch gibt es noch einige anderen speziellen Typ von Knoten: Inputknoten. Diese wirken als Startpunkt des Graphen, bei welchem die Ursprungswerte eingespielen werden. Sogenannte **placeholder** dienen beispielsweise dazu, die Features ins Modell einzufügen. Hingegen können **variable**-Inputknoten genutzt werden, um Modellparameter einzuführen. Diese

²TF besitzt mehrere Frontends, welche in den verschiedensten Programmiersprachen geschrieben sind. Wir werden jedoch nur das Python-Frontend betrachten, da es am besten von TF unterstützt ist.

Modellparameter-Tensoren sind im Gegensatz zu den vorgängig beschriebenen Tensoren “mutable”. Ihre Werte bleiben über mehrere Graphenausführungen hinweg erhalten und können auch beim Training modifiziert werden.

5.1.2.2 Operationen

Die Werte der Inputknoten gelangen über die Pfade zu den nächsten Knoten. Dort werden diese durch verschiedene **Operationen** weiterverrechnet. Durch das Definieren einer Operation wird automatisch ein neuer Knoten erstellt, in welchem das Resultat landet. Auch die benötigten Pfade werden von selbst vernetzt. Eine Operation besitzt jeweils einen Namen und repräsentiert eine abstrakte Berechnung. Sie besitzt jeweils null oder mehr Tensor-Objekte als Input, wie auch als Output.

TensorFlow offeriert eine Vielzahl von vordefinierten Operationen, sodass eine eigenständige Definition von Operationen meistens nicht notwendig ist. Ein Beispiel für eine vordefinierte Operation ist die `matmul(a, b)`-Operation. Sie stellt eine Matrixmultiplikation zwischen dem Knoten `a` und dem Knoten `b` dar. Das Resultat kann dann in einer neuen Knotenvariable `c` gespeichert werden.

Um den eigentlichen Graph zu erstellen, werden zu Beginn einige Inputknoten definiert. Diese werden durch die vordefinierten Operationen abgebildet. Die jeweiligen Resultate können weiter verrechnet werden bis dann der gewünschte Graph entsteht.

In Abbildung 5.3 ist ein Beispieldiagramm visualisiert. Dieser weist dasselbe Verhalten auf wie ein künstliches Neuron mit drei Inputs. In rot sind die `Placeholder`-Tensoren und in blau die `Variable`-Tensoren dargestellt.

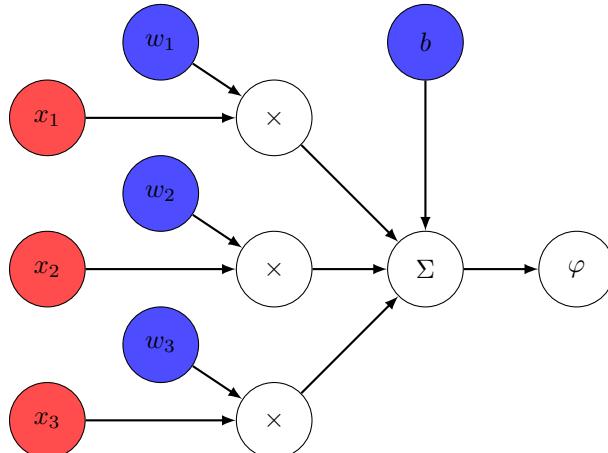


Abbildung 5.3: Beispieldiagramm, welches sich wie ein künstliches Neuron verhält

5.1.3 Ausführung

5.1.3.1 Session

Als Schnittstelle zwischen Frontend und Backend fungiert die sogenannte **Session**. Sie leitet die Informationen zum Graphen an das Backend weiter, damit er dort performant ausgeführt werden kann. Die Kernaufgabe einer Session besteht darin, den Graphen mehrfach auszuführen. Diese Ausführungen repräsentieren die Vorwärtspropagierung. Dabei kann der Benutzer die Knoten angeben, deren Werte er nach der Ausführung ausgewiesen haben möchte. Die Ausführungsfunktion erwartet als Argumente diejenigen Werte, mit denen die Inputknoten gespeist werden sollen.

5.1.3.2 Optimizer

Bevor das definierte Modell brauchbare Resultate bei der Graphenausführung liefern kann, muss es trainiert werden. Dafür kommt ein sogenannter **Optimizer** zum Einsatz. Dieser verkörpert das gewählte Optimierungsverfahren, welches zur Minimierung der Kostenfunktion verwendet wird. Es gibt verschiedene Optimizer, welche unterschiedliche Konvergenzgeschwindigkeiten und Präzisionsgrade aufweisen. Im Rahmen dieser Arbeit wurde jedoch nur das SGD-Verfahren behandelt, sodass ausschließlich dieser Optimizer hier Anwendung findet.

Beim Optimierungsverfahren muss TF die partiellen Ableitungen der Kostenfunktion bezüglich den Modellparametern bestimmen. Hierfür macht sich die Verwendung von Computational Graphs ausgezahlt, da mithilfe von ihnen der Kostengradient automatisiert bestimmt werden kann (siehe Anhang (B)).

5.1.4 Tensorboard

TensorBoard ist ein externes Zusatzprogramm zu TensorFlow. Mit dessen Hilfe können Daten zum Modell-Graph und zum Trainingsprozess erfasst und visualisiert werden. Wertvoll ist vor allem, dass die Kostenfunktionsentwicklung betrachtet werden kann.

TensorBoard ist als Webserver geschrieben und kann über einen Webbrowser benutzt werden. TensorBoard ist als Callback in TensorFlow zu aktivieren, damit die nötigen Daten während der Graphausführung gesammelt werden.

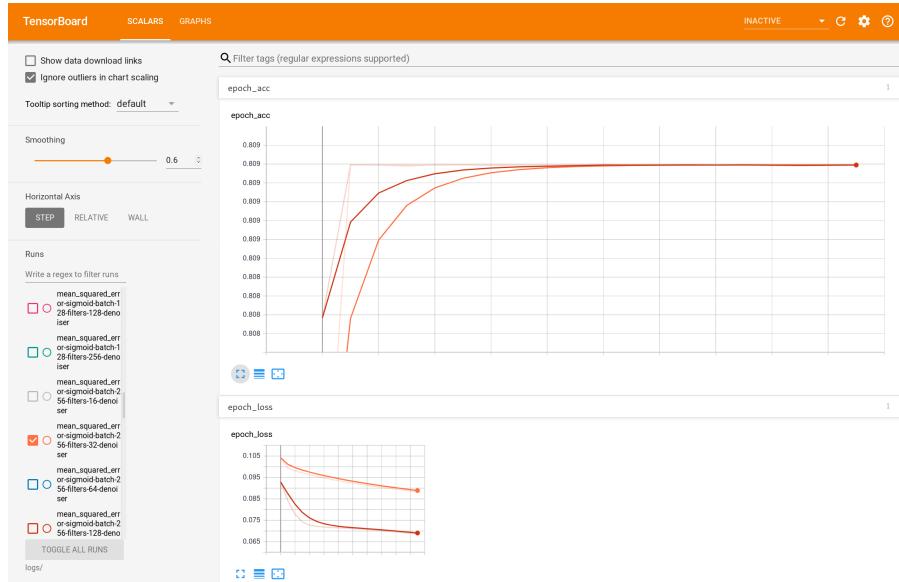


Abbildung 5.4: das TensorBoard-Webinterface

5.1.4.1 Hyperparameter einstellen

Der wichtigste Aspekt von TensorBoard ist, dass die Trainingsfortschritte visualisiert werden. Dadurch können die verschiedenen Funktionsgraphen der Kostenfunktion analysiert werden. Es ist sogar möglich, die Kostengraphen mehrerer Modelle gleichzeitig anzuzeigen und zu vergleichen. Somit kann man verschiedene Modelle mit unterschiedlichen Hyperparametern gegenüberstellen und somit die besten Hyperparameter identifizieren. Dieses Verfahren gelangt in Kapitel sechs zur Anwendung. Ebenfalls wird die beanspruchte Trainingszeit angezeigt und so beurteilbar gemacht.

In der unten stehenden Abbildung ist ein Graph von TensorBoard dargestellt, welcher die Entwicklung der Kostenfunktion für den Testdatensatz verschiedener Modelle anzeigt. In diesem Beispiel hätte die unterste orange Kurve am besten abgeschnitten. Dieses Modell hat die kleinsten Kosten verursacht. Jedoch hat dieses auch die längste Trainingszeit beansprucht.

5.2 Keras

Die Erstellung von KNNs in TensorFlow erweist sich als vergleichsweise aufwendig, weil jede KNN-Schicht manuell zu konstruieren ist. Aus diesem Grund soll nun die Funktionsweise von Keras behandelt werden. **Keras** ist eine High-Level API, welche verschiedene KNN-Schichten vorkonfiguriert zur Verfügung stellt. Somit müssen diese nicht mehr manuell erstellt werden, was allfällige Fehler vermeidet.

Keras ist ein Deep-Learning Framework, geschrieben in Python. Es wurde entwickelt, um eine einheitliche Schnittstelle für verschiedene Backends, wie TensorFlow, Microsoft Cognitive Toolkit und Theano zu bieten. Seit der TF Version 1.4 ist Keras ein fester Bestandteil der TensorFlow-Core-API. Keras ermöglicht ein benutzerfreundliches Definieren von KNNs, welche dann vereinfacht trainiert werden können.

Quellen: (27) (26)

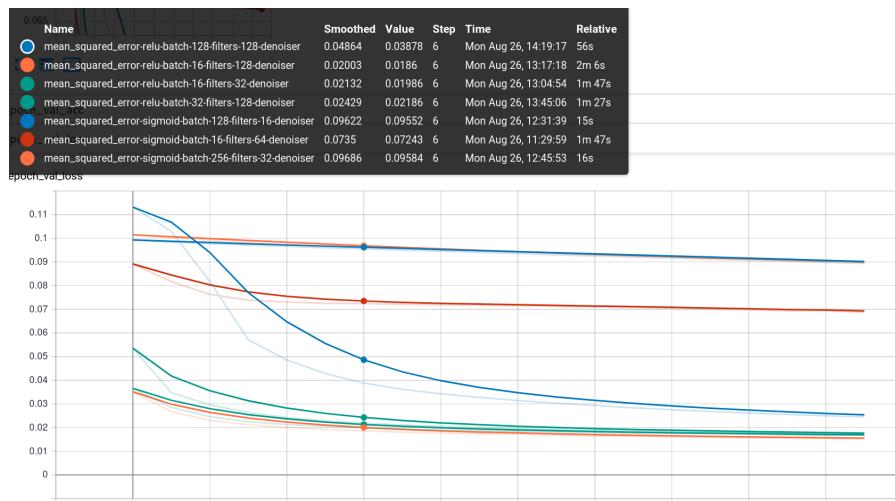


Abbildung 5.5: Trainingsvisualisierungen in TensorBoard

5.2.1 Sequential-Modell

Keras ist keine eigenständige Programmbibliothek, sondern baut auf TensorFlow auf und dient als High-level Schnittstelle. Es verwendet die verschiedenen Konzepte von TF, wie Graphen und Operationen, um eigene KNN-Bestandteile zu implementieren. Dadurch ist es für den Benutzer deutlich einfacher, ein Modell aus diesem Baukasten zu konstruieren.

Die gängigste Vorgehensweise besteht darin, ein sogenanntes **Sequential-Modell** zu bauen. Dies ist ein Modell, welches einen linearen Stapel von verschiedenen KNN-Schichten beinhaltet. Man kann dabei aus den vordefinierten Schichten wählen. Im folgenden werden diese Schichten näher betrachtet.

5.2.2 Schichten

Die Definition des Graphen in Keras erfolgt, indem die verschiedenen Schichten des gewünschten Modells in Variablen gespeichert werden. Um die Schichten miteinander zu verbinden, erfolgt eine Referenzierung auf die vorherige Schicht. Sobald alle Schichten deklariert wurden, wird eine Modellvariable definiert, welche die erste und die letzte Schicht als Argumente erwartet.

Im folgenden werden die relevanten Schichtentypen von Keras dargelegt.

Vorgängig sei darauf hingewiesen, dass Keras standardmäßig ein anderes Vorgehen zur Initialisierung der Modellparameter verwendet, als in der Theorie (siehe Sektion (2.5.1)) erklärt wurde. Keras benutzt nämlich eine Glorot-Initialisierung mit einer uniformen Verteilung. Dies steht im Gegensatz zur erklärten Theorie, wo eine Normalverteilung verwendet wurde. Aus diesem Grund muss bei jeder Schicht explizit spezifiziert werden, dass eine Glorot-Initialisierung mit Normalverteilung angewendet werden soll.

Input-Schicht

Die Input-Schicht enthält als Platzhalter-Schicht alle Features. Sie besitzt keine weitere Funktionalität. Mit folgendem Code kann sie definiert werden, dabei muss lediglich die Form des Featuretensors angegeben werden:

```
tf.keras.Input(shape=<Form>)
```

Dense-Schicht

Die erste richtige KNN-Schicht ist die Fully-Connected-Schicht. Sie wird in Keras als **Dense** bezeichnet. Sie ist wie in der Theorie erklärt, diejenige Schicht, welche aus Neuronen besteht und "dicht" zu jedem Neuron der nächsten Schicht verbunden ist. Als Argumente erwartet die Schicht die Anzahl Neuronen, aus welcher sie besteht, die Aktivierungsfunktion und die vorherige Schicht. Wie vorhin erwähnt überschreiben wir den Standard-Initialisierer mit dem '`'glorot_normal'`-Initialisierer. Ebenfalls wird die vorherige Schicht referenziert, um die Verbindung zwischen den Schichten herzustellen.

```
tf.keras.layers.Dense(<Anzahl Neuronen>,
 activation=<Aktivierungsfunktion>, kernel_initializer='glorot_normal')(<vorherige Schicht>)
```

Conv2D-Schicht

Die Convolutional-Schicht eines CNNs wird in Keras mit **Conv2D** bezeichnet. Sie verhält sich, wie in der Theorie erklärt. Zu ihrer Definition gehört die Anzahl Filter c , die Filtergrösse f , der Stride s , das Padding p und die Aktivierungsfunktion φ . Die Tensoren, welche von einer CNN-Schicht verarbeitet werden sollen, müssen folgendes Format ausweisen: (Anzahl Samples, Anzahl Farbkomponenten, Bildhöhe, Bildbreite).

```
tf.keras.layers.Conv2D(filters=<Anzahl Filter>, kernel_size=<Filtergrösse>,
 strides=<>, padding=<>, activation=<>, kernel_initializer='glorot_normal')(<vorherige Schicht>)
```

MaxPool2D-Schicht

Die **MaxPool2D**-Schicht in Keras ist diejenige Schicht, welche das MaxPooling vornimmt. Um sie zu definieren, muss die Grösse des Feldes f festgelegt werden, welches zu einem Element zusammengefasst werden soll. Des Weiteren müssen der Stride s und das Padding p spezifiziert werden.

```
tf.keras.layers.MaxPool2D(pool_size=<Feldgrösse>, strides=<>, padding=<>)(<vorherige Schicht>)
```

UpSampling2D-Schicht

Die UpSampling-Schicht, als Gegenstück zur Pooling-Schicht, heisst in Keras **UpSampling2D**. Zur Definition muss die Grösse f des neuen Feldes angegeben werden, welches die interpolierten Werte enthält. Zudem muss noch die gewählte Interpolationsmethode gewählt werden. Hier soll nur die Nächste-Nachbar-Interpolation zur Anwendung gelangen.

```
tf.keras.layers.UpSampling2D(size=<>, interpolation=<>)(<vorherige Schicht>)
```

Quellen: (27) (26)

5.2.3 Training und Evaluierung

5.2.3.1 Compile

Nachdem das Modell definiert wurde, muss es kompiliert werden. Dies geschieht mit der Methode **compile**, welche als Argumente den Optimizer und die Kostenfunktion erwartet.

```
model.compile(optimizer=<>, loss=<Kostenfunktion>)
```

Das Kompilieren bildet dann aus den gegebenen Schichten den eigentlichen Computational Graph.

Quellen: (27) (26)

5.2.3.2 Fit

Nach der Kompilierung kann das Modell trainiert werden. Dies wird durch einen einzigen Methodenaufruf durchgeführt. Zu diesem Zweck wird die **fit** Methode aufgerufen. Sie erwartet als Argumente ein Array aller Inputdaten, ein Array aller Labels, die Anzahl Epochen und die Minibatch-Grösse. Mit der Option **shuffle** kann zudem spezifiziert werden, ob Keras vor dem Training den Datensatz durchmischen soll, um eine unerwünschte Mustererkennung innerhalb der Datenabfolge zu vermeiden. Ebenfalls kann auch ein Testdatensatz angegeben werden.

```
model.fit(x=<inputs>, y=<labels>, epochs=<>, batch_size=<>, shuffle=<>,
           validation_data=(test_inputs, test_labels))
```

Quellen: (27) (26)

5.2.3.3 Predict

Sobald das Modell trainiert wurde, können damit Vorhersagen zu neuen Daten angestellt werden. Dazu diesem Zweck dient die `Predict`-Methode. Diese erwartet lediglich ein Array mit Inputs, zu welchem sie Vorhersagen erstellen soll.

```
result = model.predict(<Daten>)
```

Quellen: (27) (26)

Kapitel 6

Entwicklung eines Denoising-Autoencoders

In diesem Kapitel soll die erarbeitete Theorie zum Maschinellen Lernen und zu TensorFlow sowie Keras zusammengeführt werden, um ein umfassendes Anwendungsbeispiel zu programmieren. Es wird darauf eingegangen, welche Schritte die Entwicklung eines solchen Modells umfasst und welches Vorgehen idealtypischerweise zu wählen ist.

6.1 Das konkrete Modell

Das konkrete Modell, welches wir entwickeln werden, ist ein Convolutional-Denoising-Autoencoder. Dieser soll dazu verwendet werden, um das Bildrauschen von Bildern zu eliminieren, und sie so wieder erkennbar zu machen. Als Bilder wird ein Datensatz von handgeschriebenen Ziffern verwenden: den MNIST-Datensatz.

Das Modell besteht aus drei Typen von Schichten: Conv2D-, Pool2D- und UpSampling2D-Schichten. Die Convolutional-Schicht bildet mit einer der beiden anderen Schichten ein Paar. Nur die Convolutional-Schicht des Flaschenhalses ist allein in der Mitte positioniert. Von den jeweiligen Paaren liegen jeweils zwei vor und hinter dem Flaschenhals. Der Aufbau ist spiegelsymmetrisch zum Flaschenhals. Dabei werden die Pooling-Schichten des Encoders mit UpSampling-Schichten im Decoder ersetzt. Die Inputschicht dient lediglich als Platzhalter.

In Abbildung (6.1) ist eine Illustration des beschriebenen Modells zu sehen. Die einzelne Input-Schicht ist in rot, die Conv2D-Schichten sind in blau, die Pool2D-Schichten in gelb und die UpSampling2D-Schichten in orange dargestellt.

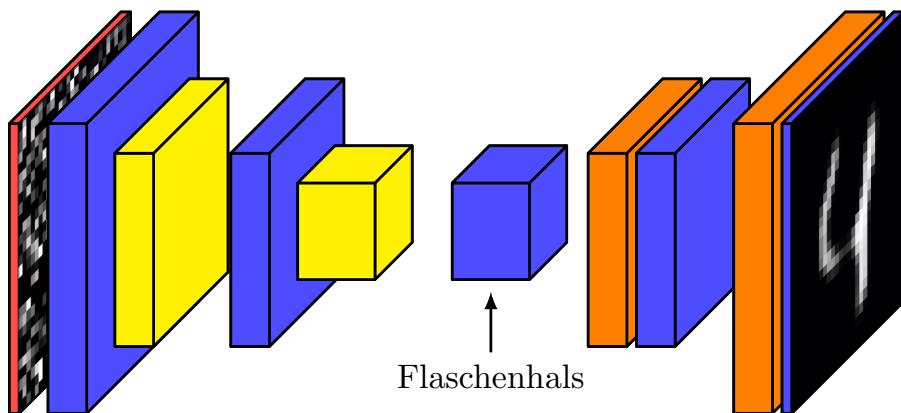


Abbildung 6.1: Schema des beschriebenen Modells (Tensoren sind nicht massstabsgetreu)

6.1.1 Daten

Als Trainingsdaten wird der **MNIST**-Datensatz verwendet. Es handelt sich dabei um den wohl bekanntesten Datensatz für beispielhaftes Maschinelles Lernen. Er besteht aus schwarz-weiss Bildern von handgeschriebenen Ziffern. Der Datensatz würde eigentlich auch Labels zu den Ziffern enthalten, um das Modell beispielsweise für

Ziffernerkennung trainieren zu können. Jedoch ist dies hier nicht notwendig, weil ein Autoencoder entwickelt werden soll, dessen Labels den Inputs entsprechen.

Der Datensatz besteht aus einem Trainingsdatensatz von 60'000 Samples und einem Testdatensatz von 10'000 Samples. Alle Ziffern sind bereits korrekt formatiert, da ihre Grösse auf 28×28 Pixel normalisiert wurde und sie im Bild zentriert sind. Ein Auszug an MNIST-Ziffern ist in Abbildung (6.2) zu sehen.



Abbildung 6.2: ein Auszug an MNIST-Bildern (invertierte Farben) (28)

Quellen: (29)

6.2 Setup

Zunächst ist es wichtig, das Entwicklungsumfeld richtig zu konfigurieren. Das bedeutet konkret, dass Python zusammen mit den verschiedenen Programmabibliotheken installiert werden muss. Die Installationsschritte werden in dieser Arbeit für eine arch-basierte Linuxdistribution erklärt, welche den **Pacman**-Package-Manager verwendet. Falls der Leser ein anderes Betriebssystem verwenden möchte, ist auf die offizielle TensorFlow Website für die Installationsschritte zu verweisen (siehe QR-Code).



Abbildung 6.3: QR-Code
für Installationsanweisun-
gen

Python3

Damit TensorFlow und Keras verwendet werden können, bedingt dies, dass Python3 installiert sein muss. Folgender Befehl muss in der Kommandozeile ausgeführt werden, um die Installation durchzuführen.

```
sudo pacman -S python
```

TensorFlow

Für die Installation von TensorFlow muss unter Arch Linux das entsprechende Package installiert werden. Falls der Leser über eine Nvidia-Grafikkarte verfügt, welche eine “Compute Capability” von mehr als 3.5 besitzt, kann er von der GPU-Hardwarebeschleunigung Gebrauch machen. Somit ist die CUDA-Version von TF mithilfe des folgenden Kommandos zu installieren:

```
sudo pacman -S python-tensorflow-cuda
```

Pacman installiert nun automatisch zuerst alle Programmabhangigkeiten, wie CUDA, cuDNN und das TensorFlow-Backend, bevor das eigentliche Python-TensorFlow installiert wird.

Falls keine geeignete Grafikkarte vorliegt, ist es ebenso möglich, die normale TF-Version ohne Unterstützung für die GPU zu verwenden. Dies kann durch nachstehendes Kommando vorgenommen werden:

```
sudo pacman -S python-tensorflow
```

Dabei muss Keras nicht explizit installiert werden, da es bereits in TensorFlow implementiert ist.

Python-Module

Für das Python-Programm werden zwei Packages benötigt, welche nicht in der Standardbibliothek von Python enthalten sind:

- NumPy: Ein Package, welches verschiedene mathematische Konzepte implementiert; vor allem Vektor- und Matrix-Arithmetik.
- Matplotlib: Ein Package zum Erstellen von Plots und Grafiken.

Man installiert beide mit folgenden Kommandos:

```
sudo pacman -S python-numpy
sudo pacman -S python-matplotlib
```

6.3 Entwicklung

Nun kann die eigentliche Programmierung beginnen.

6.3.1 Testprogramm

Zunächst wird ein kleines Testprogramm geschrieben, welches überprüft, ob alle Programmabhangigkeiten korrekt installiert sind und verwendet werden können. In den ersten Zeilen des Pythonprogramms werden die Importstatements geschrieben, um NumPy, Matplotlib und TensorFlow verfügbar zu machen. Keras muss wie gesagt nicht explizit geladen werden, da es in TensorFlow enthalten ist. Nun können die Versionsnummern der verschiedenen Programmabibliotheken in die Konsole geschrieben werden, um zu überprüfen, ob alles richtig konfiguriert ist.

```
1 import numpy as np # NumPy wird importiert
2 import matplotlib as mpl # Matplotlib wird importiert
3 import tensorflow as tf # TensorFlow wird importiert
4
5 print(np.__version__) # Schreibt die NumPy-Version nach stdout.
6 print(mpl.__version__) # Schreibt die Matplotlib-Version nach stdout.
7 print(tf.__version__) # Schreibt die TensorFlow-Version nach stdout.
```

Falls der Output folgenden Charakter hat (die Versionsnummern müssen nicht die gleichen sein) und keine Fehlermeldungen erfolgen, sollte alles funktionieren.

```
1 1.17.0
2 3.1.1
3 1.14.0
```

6.3.2 Trainingsdaten

Nun wurde mit dem kleinen Testprogramm verifiziert, dass alle Programmabhängigkeiten funktionieren. Auf dieser Basis können wir mit dem eigentlichen Programm beginnen.

In den folgenden Ausschnitten wird der Code laufend weiter ausgebaut. Auf diese Weise zeigt sich, was er bewirkt. Das eigentliche Programm startet mit folgenden Importstatements:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import tensorflow as tf
```

6.3.2.1 Laden des MNIST-Datensatzes

Der erste Schritt besteht darin, die Datensätze zu laden. Wie bereits erwähnt, wird dabei vom MNIST-Datensatz Gebrauch gemacht. Da dieser stark verbreitet ist, existiert eine Funktion in Keras, welche automatisch diese Daten herunterlädt und sie als NumPy-Arrays zur Verfügung stellt. Die Funktion gibt die verschiedenen Komponenten der Daten in folgendem Format zurück (`x_train`, `y_train`), (`x_test`, `y_test`). Da nur Interesse an den Features `x` besteht, werden die überflüssigen Labels `y` mithilfe der Wegwerf-Variable `_` verworfen.

```
1 (x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
```

Nun sind die Features für das Training in der Variable `x_train` und die Features des Trainingsdatensatzes in der Variable `x_test` gespeichert.

6.3.2.2 Formatieren der Daten

In einem nächsten Schritt müssen die Daten transformiert werden, damit sie die richtige Form für das Modell aufweisen. Wie bereits erwähnt, ist der MNIST-Datensatz unkompliziert verwendbar, da alle Bilder die gleichen Masse aufweisen und die Ziffern zentriert sind.

Jedoch sind die Grauwerte zu normalisieren, denn im Moment liegen sie noch im Intervall [0, 255] und sind vom Typ Integer. Das Modell kann am besten mit Kommazahlen, welche im Intervall [0, 1] liegen, umgehen. Um diese Anpassung vorzunehmen, kann folgender Code angefügt werden:

```
1 x_train = x_train.astype('float32') / 255.0 # Normalisierung
2 x_test = x_test.astype('float32') / 255.0 # Normalisierung
```

Des Weiteren werden im Modell Conv2D-Schichten verwendet. Diese in Keras implementierten Conv2D-Schichten erwarten Inputs in der Form (m, w, h, c) , wobei m die Anzahl der Bilder ist, w die Bildbreite, h die Bildhöhe und c die Anzahl der Farbkomponenten. Die Bildbreite w und -höhe h von 28 Pixeln wird beibehalten, wie auch die Anzahl Farbkomponenten $c = 1$. Die Anzahl Bilder m lässt sich aus der Länge des Arrays entnehmen `len(x_train)`. Mithilfe vom NumPy lässt sich das Array wie folgt umformen.

```
1 x_test = np.reshape(x_test, (len(x_test), 28, 28, 1)) # neue Form: (60'000, 28, 28, 1)
2 x_train = np.reshape(x_train, (len(x_train), 28, 28, 1)) # neue Form: (10'000, 28, 28, 1)
```

Die Daten besitzen nun die richtige Form für das Modell.

6.3.2.3 Generieren der verrauschten Bilder

Als Input für das Modell werden nicht die normalen MNIST-Bilder verwendet, sondern eine verrauschte Variante von diesen. Sie werden generiert, indem ein additives Gauss'sches Rauschen auf sie angewendet wird.

Zuerst wird eine Matrix $\mathbf{R} \in \mathbb{R}^{28 \times 28 \times 1}$ erstellt, welche die gleiche Form wie die Bilder besitzt. Diese Matrix wird mit Zufallswerten gefüllt. Für die Zufallswerte wird eine normalisierte Gauss'sche Normalverteilung $\mathcal{N}(\mu = 0, \sigma^2 = 1)$ mit Erwartungswert $\mu = 0$ und Varianz $\sigma^2 = 1$ verwendet. Da jedes Bild eine eigene Rauschmatrix verlangt, wird zu diesem Zweck eine Liste $(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_m)$ der Länge m an Rauschmatrizen erzeugt. Dafür kann die NumPy Funktion `np.random.normal(loc=<μ>, scale=<σ²>, size=<Form>)` verwendet werden. Die somit erhaltenen Rauschmatrizen werden dann mit einer Rauschkonstante `noise_factor` multipliziert und

anschliessend wird das Produkt auf die MNIST-Bilder addiert. Vorerst wird der `noise_factor = 0.5` gewählt. Nach dem Hinzufügen der Rauschwerte müssen die Grauwerte noch auf das Intervall $[0, 1]$ zurecht gestutzt werden. Auch hierfür kann eine NumPy-Funktion `np.clip(var, min, max)` angewendet werden. Diese Schritte werden sowohl mit dem Trainingsdatensatz, als auch mit dem Testdatensatz vollzogen. Somit lautet der Code:

```

1 noise_factor = 0.5
2
3 # für x_train
4 noise_matrices = np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
5 noise_matrices *= noise_factor
6 x_train_noisy = x_train + noise_matrices
7 x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
8
9 # für x_test in Kurzfassung
10 x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
11 x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)

```

Mithilfe der Matplotlib kann ein Blick auf die verrauschten Bilder im Vergleich zu den Originalbildern geworfen werden. Dafür wird ein Plot erstellt, welcher jeweils 10 Bilder beider Arrays zeigt.

Für diesen Zweck wird eine `pyplot.figure` erstellt. Innerhalb dieser werden jeweils 10 `subplots` definiert, sowohl für die verrauschten Bilder, als auch für die Originale. Bevor die Bilder angezeigt werden können, müssen sie in die richtige Form für die Matplotlib gebracht werden. Sie sollen die Form (28×28) aufweisen. Ausserdem ist der Plot als schwarz-weiss Grafik zu spezifizieren.

```

1 n = 10 # jeweils 10 Bilder
2 plt.figure()
3 for i in range(n):
4     # Originalbilder
5     ax = plt.subplot(2, n, 1+i)
6     img_clean = x_test[i].reshape(28, 28)
7     plt.imshow(img_clean)
8     plt.gray()
9
10    # verrauschte Bilder
11    ax = plt.subplot(2, n, 1+n+i)
12    img_noisy = x_test_noisy[i].reshape(28, 28)
13    plt.imshow(img_noisy)
14    plt.gray()
15 plt.show()

```

Das Ergebnis ist in Abbildung (6.4) sichtbar.

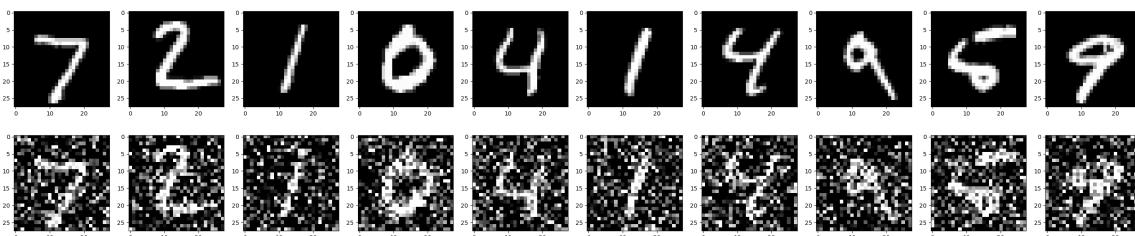


Abbildung 6.4: Verrauschte Bilder neben den Originalbildern

Die Daten liegen nun vollständig und im richtigen Format vor. Nun kann zur Definition des Modells übergegangen werden.

6.3.3 Modell definieren

Mithilfe von Keras kann das Modell eines Convolutional-Denoising-Autoencoder definiert werden. Dies geschieht nach dem Vorbild der Theorie. Wie bereits erklärt umfasst die Topologie eines CNNs die verschiedensten Hyper-

parameter. Für diese werden der Einfachheit halber zunächst willkürliche Werte verwendet. Zu einem späteren Zeitpunkt werden die passenden Hyperparameter durch iteratives Ausprobieren ermittelt.

Das Modell beginnt mit der Inputschicht $l = 0$, welche ein Tensor ist, der die Inputwerte enthält. Die Schicht hat logischerweise die gleiche Form, wie ein MNIST-Bild.

```
1 input_data = tf.keras.Input(shape=(28, 28, 1))
```

Nach dieser Inputschicht folgt der Encoder des Autoencoders. Dieser besteht sowohl aus Convolutional-Schichten, wie auch aus Pooling-Schichten, welche sich abwechseln. Für alle Schichten werden vorerst unbegründete Hyperparameter gewählt. Für die Convolutional-Schichten wird eine Filtergrösse $f^l = 3$, eine Anzahl Filter von $c^l = 32$, ein Stride $s^l = 1$, Same-Padding und die ReLU-Aktivierungsfunktion $\varphi = \varphi^{\text{ReLU}}$ verwendet. Da Keras standardmässig die Initialisierung der Modellparameter mit einer uniformen Verteilung vornimmt, muss der `kernel_initializer` explizit auf `'glorot_normal'` gesetzt werden. Dadurch findet die Initialisierung analog zu Sektion 2.5.1 statt.

Der Code für die Definition einer solchen Convolutional-Schicht lautet:

```
1 conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3),
2     strides=(1, 1), padding='same', activation='relu', kernel_initializer='glorot_normal')
```

Für die Pooling-Schichten wird analog eine Feldgrösse $f = 2$, ein Stride $s = 2$ und Same-Padding gewählt. Dies bewirkt, dass jedes (2×2) -Feld zu einem einzigen Element zusammengefasst wird.

Der Code für eine dieser Pooling-Schichten lautet:

```
1 pool_layer = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='same')
```

Für den Decoder werden die gleichen Convolutional-Schichten wie für den Encoder verwendet. Anstelle von Pooling-Schichten verwendet er UpSampling-Schichten, welche jeweils nach einer Convolutional-Schicht folgen. Für die UpSampling-Schichten wurde eine Feldgrösse $f = 2$ gewählt. Somit wird ein einziger Pixel auf ein (2×2) -Felder hochskaliert. Als Interpolationsmethode wird der Nächste-Nachbar-Algorithmus verwendet. Die beschriebene UpSampling-Schicht wird folgendermassen definiert:

```
1 upsampling_layer = tf.keras.layers.UpSampling2D(size=(2, 2), interpolation='nearest')
```

Nun folgt die Deklarierung aller Schichten und deren Verknüpfung.

Das Modell weist einen Flaschenhals auf, welcher zwei Convolutional-Schichten tief ist. Das bedeutet, dass der Encoder aus zwei Conv-Pool-Paaren besteht. Danach folgt der einschichtige Flaschenhals. Im Anschluss daran folgt der Decoder, der zwei Conv-UpSampling-Paare umfasst. Um die jeweils neue Schicht zur vorherigen Schicht zu verbinden, gibt man die vorherige in Klammern am Ende des Statements an.

Der ganze Graph wird durch folgenden Code gebildet:

```
1 # Encoder
2 input_data = tf.keras.Input(shape=(28, 28, 1))
3 econv0 = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same',
4     activation='relu', kernel_initializer='glorot_normal')(input_data)
5 emaxpool0 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='same')(econv0)
6 econv1 = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same',
7     activation='relu', kernel_initializer='glorot_normal')(emaxpool0)
8 emaxpool1 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='same')(econv1)
9
10 # Flaschenhals der Form (7, 7, 32)
11 encoded = emaxpool1
12
13 # Decoder
14 dconv0 = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same',
15     activation='relu', kernel_initializer='glorot_normal')(encoded)
16 dupsample0 = tf.keras.layers.UpSampling2D(size=(2, 2), interpolation='nearest')(dconv0)
17 dconv1 = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same',
18     activation='relu', kernel_initializer='glorot_normal')(dupsample0)
```

```

19 dupsample1 = tf.keras.layers.UpSampling2D(size=(2, 2), interpolation='nearest')(dconv1)
20 dconv2 = tf.keras.layers.Conv2D(filters=1, kernel_size=(3, 3), strides=(1, 1), padding='same',
21     activation='sigmoid', kernel_initializer='glorot_normal')(dupsample1)
22 decoded = dconv2

```

Auffallend im Code ist, dass der Decoder nicht etwa zwei Convolutional-Layers besitzt, sondern sogar drei. Die letzte Schicht wird benötigt, damit die Daten wieder die richtige Formatierung aufweisen. Das heisst einerseits, dass die Outputs wieder in der Ursprungsform (28×28) vorliegen müssen. Andererseits sollen sich die Outputs wieder im Intervall $[0, 1]$ befinden. Mithilfe der Sigmoid-Aktivierungsfunktion φ^{sig} werden sie genau auf dieses Intervall abgebildet. Für alle anderen Schichten wurde die ReLU-Aktivierungsfunktion gewählt.

Nun muss noch eine Modellvariable deklariert werden. Als Argumente erwartet diese die Inputschicht `input_data` und die Outputschicht `decoded`. Danach ist das Modell zu kompilieren. Bei der Kompilierung kann das Optimierungsverfahren und die Kostenfunktion gewählt werden. Für die Optimierung wird das Stochastische Gradientenverfahren SGD verwendet. Als Kostenfunktion wird der Mittlere-Quadratische-Fehler C_{MSE} gewählt.

```

1 autoencoder = tf.keras.Model(input_data, decoded)
2 autoencoder.compile(optimizer='sgd', loss='mean_squared_error') # SGD und MSE

```

Mithilfe von `tf.keras.Model.summary` kann man eine Zusammenfassung des Modells in Textform erhalten. So können Informationen bezüglich der Form der verschiedenen Schichten abgerufen werden, um zu überprüfen, ob alles stimmig ist. Nachfolgende Summary gilt für unser Modell:

```

1 -----
2 Layer (type)          Output Shape         Param #
3 -----
4 input_1 (InputLayer)   [(None, 28, 28, 1)]   0
5 -----
6 conv2d (Conv2D)        (None, 28, 28, 32)    320
7 -----
8 max_pooling2d (MaxPooling2D) (None, 14, 14, 32) 0
9 -----
10 conv2d_1 (Conv2D)      (None, 14, 14, 32)    9248
11 -----
12 max_pooling2d_1 (MaxPooling2D) (None, 7, 7, 32) 0
13 -----
14 conv2d_2 (Conv2D)      (None, 7, 7, 32)    9248
15 -----
16 up_sampling2d (UpSampling2D) (None, 14, 14, 32) 0
17 -----
18 conv2d_3 (Conv2D)      (None, 14, 14, 32)    9248
19 -----
20 up_sampling2d_1 (UpSampling2D) (None, 28, 28, 32) 0
21 -----
22 conv2d_4 (Conv2D)      (None, 28, 28, 1)     289
23 -----
24 Total params: 28,353
25 Trainable params: 28,353
26 Non-trainable params: 0
27 -----

```

Es zeigt sich, dass das Modell gesamthaft circa 28'000 Parameter umfasst, welche zu trainieren sind.

Quelle: (30)

6.3.4 Modell trainieren

Das definierte Modell ist nun zu trainieren. Mit Keras kann dies durch einen einzigen Funktionsaufruf erfolgen. Diese Funktion erwartet einige Argumente. Zuerst sind die Features und Labels anzugeben. Die Features wurden in der Variable `x_train_noisy` gespeichert. Die Labels sind die unverrauschten MNIST-Bilder, welche in der Variable `x_train` enthalten sind. Des Weiteren wird die Grösse eines Mini-Batches als 128 Samples spezifiziert.

Das Training erfolgt über 100 Epochen. Wichtig ist, dass die Trainingssamples vor dem Training durchmischt werden, damit das Modell nicht versucht, Muster innerhalb der Anordnung der Samples zu erlernen. Als letztes Argument wird noch der Testdatensatz angegeben.

```
1 autoencoder.fit(x=x_train_noisy, y=x_train, batch_size=128, epochs=100, shuffle=True,
2 validation_data=(x_test_noisy, x_test))
```

Sofern TensorFlow eine geeignete Nvidia-Grafikkarte vorfindet und auch die CUDA-Version von TF installiert ist, erfolgt das Training mithilfe von GPU-Hardwarebeschleunigung. Dadurch sollte das Trainings innerhalb einiger Minuten abgeschlossen sein. Während des Trainings schreibt TensorFlow Informationen zum aktuellen Fortschritt in die Kommandozeile. Diese Informationen weisen nachstehenden Charakter auf und geben Auskunft über die aktuelle Epoche sowie die Werte der Kostenfunktion. Die Kosten für den Trainingsdatensatz werden mit `loss` bezeichnet und die für den Testdatensatz mit `val_loss`.

```
1 Epoch 3/100
2 60000/60000 [=====] - 4s 60us/sample - loss: 0.0911 - val_loss: 0.0772
```

Die Kostenfunktion weist nach dem Training in Bezug auf den Testdatensatz einen Wert `val_loss` von 0.0180, auf.

Jetzt, da das Modell trainiert wurde, ist es sinnvoll, es auf dem Computer als Modelldatei abzuspeichern. So muss es nicht jedes Mal wieder neu trainiert werden, sondern kann über die Modelldatei eingelesen werden.

```
1 autoencoder.save('denoiser.model')
```

Für dieses Einlesen wird folgender Code benötigt:

```
1 autoencoder = tf.keras.models.load_model('denoiser.model')
```

Mit der `Summary`-Funktion kann man überprüfen, ob das richtige Modell geladen wurde.

6.3.5 Modell ausführen

Da das Modell nun trainiert ist, kann es für seinen eigentlichen Zweck genutzt werden. Dieser besteht darin, Bilder zu entrauschen. Dafür gelangen die Bilder aus dem Testdatensatz zur Anwendung. Auf diese Weise wird garantiert, dass diese unbekannten Daten vom Modell nicht auswendig gelernt werden konnten. Mit der Funktion `tf.keras.model.predict` erhält man alle Vorhersagen des Autoencoders zum gesamten Testdatensatz.

```
1 denoised_imgs = autoencoder.predict(x_test)
```

Mithilfe von der Matplotlib können die entrauschten Bilder mit den ursprünglichen MNIST-Bildern direkt verglichen werden. So kann mit bloßem Auge beurteilt werden, wie gut die Ergebnisse des Modells tatsächlich sind.

```
1 n = 10 # jeweils 10 Bilder
2 plt.figure()
3 for i in range(n):
4     # verrauschte Bilder
5     ax = plt.subplot(3, n, 1+i)
6     plt.imshow(x_test_noisy[i].reshape(28, 28))
7     plt.gray()
8
9     # entrauschte Bilder
10    ax = plt.subplot(3, n, 1+n+i)
11    plt.imshow(decoded_imgs[i].reshape(28, 28))
12    plt.gray()
13
14    # Original-Bilder
15    ax = plt.subplot(3, n, 1+2*n+i)
```

```

16 plt.imshow(x_test[i].reshape(28, 28))
17 plt.gray()
18 plt.show()

```

In der ersten Zeile der Grafik sind die verrauschten Bilder dargestellt. Darunter folgen die Rekonstruktionen als Ergebnis des Modells und in der dritten Zeile sind die Original-Bilder zu sehen. Wirkliche Unterschiede zwischen der zweiten und der dritten Zeile sind kaum zu erkennen.

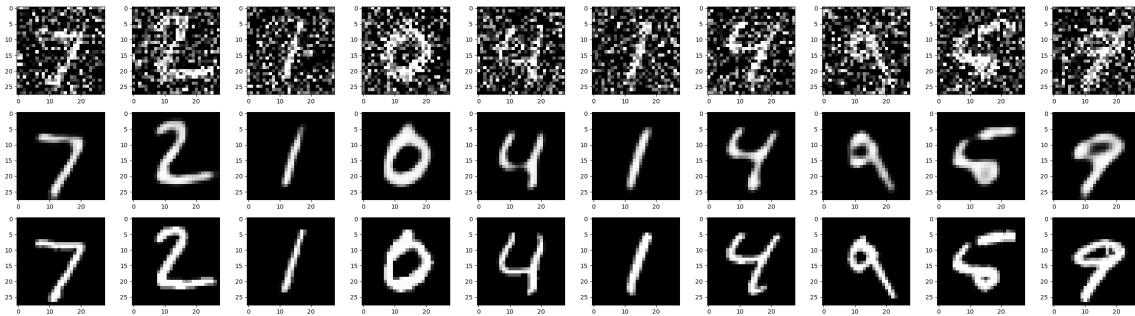


Abbildung 6.5: Verrauschte Bilder, Rekonstruktionen und Originalbilder

6.3.6 Hyperparameter einstellen

Für die Hyperparameter haben wir zunächst willkürliche Werte verwendet. Es sollen nun besser geeignete Hyperparameter bestimmt werden. Dazu müssen lediglich verschiedene Werte iterativ ausprobiert werden. Die Applikation wird modifiziert, indem verschiedene Modelle konstruiert werden, welche unterschiedliche Hyperparameter verwenden. Beim Training werden mithilfe von TensorBoard Daten zum Lernfortschritt und den Kosten der verschiedenen Modelle erfasst. Auf diese Weise kann eine Auswertung erstellt werden, um die besten Hyperparameter zu identifizieren.

Zuerst werden einige Arrays definiert, welche die zu testenden Hyperparameter beinhalten. Ein Hyperparameter ist die Anzahl Epochen. Für diesen werden wir aber nicht verschiedene Werte ausprobieren, da es offensichtlich ist, dass sich mit einer längeren Trainingszeit auch die Resultate verbessern. Alle Modelle werden für 20 Epochen trainiert.

Folgende Mini-Batch-Größen sollen geprüft werden: 16, 32, 64, 128, 256. Als zu testende Anzahl von Filtern werden definiert: 16, 32, 64, 128, 256. Für die Aktivierungsfunktionen werden Sigmoid und ReLU ausprobiert. Anstatt dass jede Schicht eine eigene Aktivierungsfunktion erhält, gelangt für das gesamte Modell eine einzige Funktion zur Anwendung. Auf diese Weise soll die Anzahl an Permutationen der Hyperparameter gering gehalten werden. Jedoch wird für die letzte Schicht immer die Sigmoid-Aktivierungsfunktion verwendet, damit die Outputs im Intervall [0, 1] liegen.

```

1 EPOCHS = 20
2
3 BATCH_SIZES = [ 16, 32, 64, 128, 256 ]
4 FILTER_NUMS = [ 16, 32, 64, 128, 256 ]
5 ACTIVATIONFUNCTIONS = [ 'sigmoid', 'relu' ]

```

Nun werden wir den gesamten Code zum Erstellen des Graphen, zur Kompilierung und zum Training des Modells in verschachtelte For-Loops verschieben, welche alle Kombinationen von Hyperparametern ausprobieren.

```

1 for activation_function in ACTIVATIONFUNCTIONS:
2     for batch_size in BATCH_SIZES:
3         for num_filters in FILTER_NUMS:
4             # ganzer Modell-Code

```

Nun muss der Code zur Definition des Modells so modifiziert werden, dass die Konstanten an den entsprechenden Stellen durch die Variablen `activation_function`, `batch_size` und `num_filters` ersetzt werden.

6.3.6.1 TensorBoard konfigurieren

Um den Lernprozess und die Leistungsfähigkeit des Modells zu analysieren, soll nun TensorBoard konfiguriert werden. Zu diesem Zweck müssen die Methoden `compile` und `fit` angepasst werden. Zuerst müssen wir in der `compile` Methode die zu erfassenden Metriken angeben. Von Interesse ist die `'accuracy'`.

```
1 autoencoder.compile(optimizer='sgd', loss='mean_squared_error', metrics=['accuracy'])
```

Nun ist eine TensorFlow-Variable zu definieren. Es ist anzugeben, wo die erfassten Daten gespeichert werden sollen. Es ist sinnvoll, den Dateien der einzelnen Modelle aussagekräftige Namen zu geben. Deshalb benennen wir die Dateien in folgendem Format: “denoiser-(Aktivierungsfunktion)-(Mini-Batch Grösse)-batches-(Anzahl Filter)-filters”.

```
1 NAME = 'denoiser-{}-{}-batches-{}-filters'.format(activation_function, batch_size, num_filters)
2 tensorboard = tf.keras.callbacks.TensorBoard(log_dir='logs/{}'.format(NAME))
```

Zuletzt muss noch die TensorBoard-Variable als `callback` in der `fit`-Methode spezifiziert werden.

```
1 autoencoder.fit(x=x_train_noisy, y=x_train, batch_size=batch_size, epochs=EPOCHS, shuffle=True,
2 validation_data=(x_test_noisy,x_test), callbacks=[tensorboard])
```

Nun kann das modifizierte Programm ausgeführt werden. Dabei werden alle spezifizierten Kombinationen an Hyperparametern durchgetestet. Gleichzeitig erzeugt das TensorBoard-Callback Logdateien im `'logs'`-Verzeichnis. Diese Dateien enthalten alle nötigen Informationen zur anschliessenden Auswertung.

6.3.6.2 TensorBoard-Analyse

Um die besten Hyperparameter mithilfe von TensorBoard zu bestimmen, muss zunächst der TensorBoard-Webserver gestartet werden. Dafür navigiert man mit der Kommandozeile in das Verzeichnis, welches die Logdateien enthält. Dort kann dann der Webserver mithilfe folgendem Kommando gestartet werden.

```
1 tensorboard --logdir .
```

Nun kann mit einem Webbrowser zur angegebenen Webadresse navigiert werden. In der Regel lautet diese: `http://localhost:6006/`. Der Graph, welcher die Kosten bezüglich des Testdatensatzes aufzeichnet, ist von besonderem Interesse. Dieser sieht folgendermassen aus:

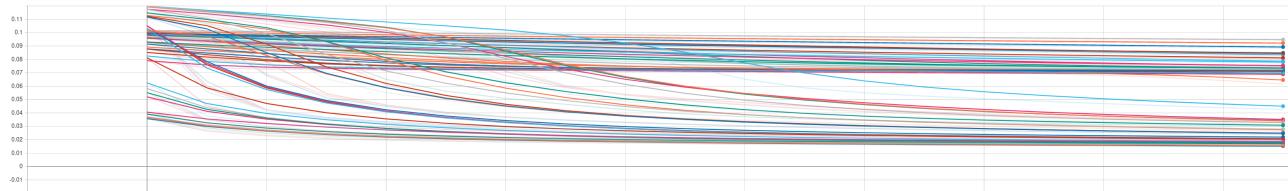


Abbildung 6.6: Kosten bezüglich dem Testdatensatz

Dank dem interaktiven Interface von TensorBoard, kann dem Graphen entnommen werden, welche Werte der Hyperparameter die besten Resultate liefern.

Die geeigneten Hyperparameter konnten jetzt identifizieren werden. Um die Leistungsfähigkeit des Modells aufzuzeigen, wird es unter Verwendung der gefundenen Hyperparameter nochmals zur Bildentrauschung eingesetzt. Das diesbezügliche Training soll wiederum 100 Epochen umfassen. Aufgrund der erwarteten höheren Leistungsfähigkeit wird in diesem Durchlauf der `noise_factor` von 0.5 (vgl. (6.3.2.3)) auf 1 gesetzt. Das Rauschentfernen wird dadurch anspruchsvoller. Die Ergebnisse sind in unten stehender Abbildung zu erkennen.

6.3.7 Diskussion des Modells

Das entwickelte Modell soll an dieser Stelle bezüglich seiner Leistungsfähigkeit diskutiert werden.

Rang	Aktivierungsfunktion	Minibatch-Grösse	Anzahl Filter	Kosten
1	ReLU	16	265	0.015
2	ReLU	16	128	0.01521
3	ReLU	16	64	0.01588
4	ReLU	16	32	0.0167
:	:	:	:	:
50	Sigmoid	256	16	0.09438

Tabelle 6.1: Rangliste der besten Modelle mit ihren Eigenschaften

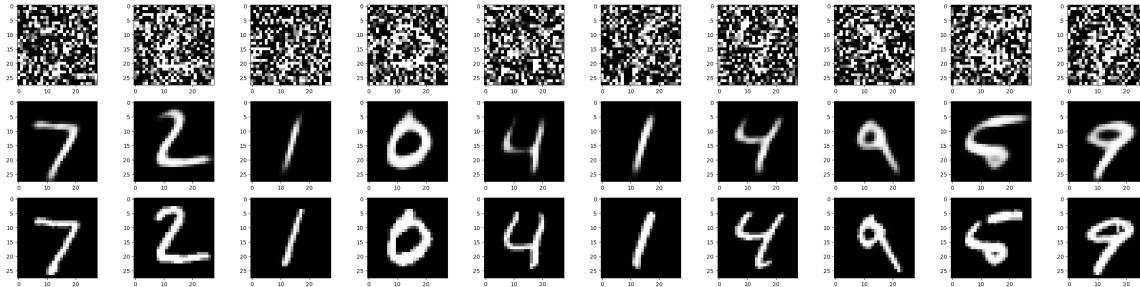


Abbildung 6.7: Sehr verrauschte Bilder, Rekonstruktionen und Originalbilder

6.3.7.1 Leistung des Modells

Die Leistung des Modells kommt am besten in der letzten Abbildung zur Geltung. Da die Hyperparameter nun optimaler gewählt wurden, ist das Modell auch in der Lage, extrem verrauschte Ziffern wieder gut erkennbar zu machen. Dem menschlichen Auge fällt es schwer, noch klare Ziffern in den verrauschten Bildern auszumachen. Das Modell hingegen kann verblüffend gut die Ziffern rekonstruieren und gänzlich von ihrem Rauschen befreien. Dies ist für ein vergleichsweise einfaches Programm eine bemerkenswerte Leistung.

6.3.7.2 Ergebnisse der Hyperparameter

Die Ergebnisse der Auswertung mit TensorBoard stimmen ziemlich gut mit meinen Erwartungen überein. Wie bereits in der Theorie geäussert, liefert die ReLU-Aktivierungsfunktion bessere Ergebnisse in CNNs als die Sigmoidfunktion. Deshalb sind die ersten Plätze der Rangliste (vgl. Tabelle (6.1)) nur mit Modellen, welche die ReLU-Funktion verwenden, besetzt. Weiter ist auffallend, dass die besten Modelle möglichst tiefe Minibatch-Größen verwenden. Dies ist naheliegend, da eine kleine Grösse mit einer höheren Präzision der Gradientenapproximation einhergeht. Somit ist der Gradientenabstieg exakter. Jedoch hat dies auch seine Schattenseite: Die Motivation hinter zur Einführung der Minibatches für SGD liegt darin, die Trainingszeit zu verkürzen. Durch die kleineren Batches nimmt die Trainingszeit jedoch erheblich zu. Eine weitere Feststellung ist, dass eine höhere Anzahl an Filtern das Modell zu besseren Resultaten führt. Jedoch ist der Einfluss dieses Hyperparameters geringer als jene der übrigen.

6.3.7.3 Verbesserungsmöglichkeiten

Es gibt konkrete Verbesserungsmöglichkeiten, welche bei der Entwicklung des Denoisers bewusst nicht umgesetzt wurden. Diese hätten womöglich für noch bessere Resultate gesorgt. Allerdings hätten sie den Umfang der vorliegenden Arbeit gesprengt.

Ein erster Ansatzpunkt hätte möglicherweise darin bestanden, mehr Werte für die Hyperparameter auszuprobieren, anstatt nur einige wenige zu testen.

Auch wäre es möglich gewesen, anderweitige Hyperparameter einzubeziehen, welche im vorliegenden Modell keine Berücksichtigung gefunden haben. So wurden beispielsweise die Topologie und die Schichtenarten gar nicht verändert. Dies hatte den praktischen Grund, dass die Anzahl Permutationen an Hyperparametern nochmals stark zugenommen und sich die Trainingsdauer deutlich verlängert hätte. Ausserdem ist es schwierig, bei einem Convolutional-Autoencoder verschiedene Topologien automatisiert zu testen. Es muss nämlich garantiert werden, dass die Inputschicht die gleichen Dimensionen wie die Outputschicht besitzt. Dies stellt sich als weitere Herausforderung dar.

Eine weitere Modifikation, welche das Modell vermutlich nochmals verbessert hätte, wäre eine alternative Kostenfunktion gewesen. So gilt beispielsweise die **Kreuzentropie-Kostenfunktion** als deutlich raffinierter als

die MSE-Kostenfunktion. Sie stellt eine Lösung für das sogenannte **Vanishing-Gradient-Problem** dar. Durch dessen Behebung kommt es nicht — wie im vorliegenden Modell — zu einer Verlangsamung des Trainings im Verlauf des Gradientenabstiegs.

Weitere Verbesserungen könnten in der Wahl eines anderen Optimierungsverfahrens als SGD liegen. Beispielsweise könnte eine SGD-Variante mit Impuls (engl.: momentum) verwendet werden. Diese Art von SGD stellt ein Verfahren dar, welches den Gradientenabstieg weniger sprunghaft macht und damit zielstrebiger zum Minimum führt.

Viele dieser Modifikationen wären durch einige wenige Codeänderungen im Programm umsetzbar. Ohne deren theoretische Behandlung sind sie jedoch nur schwer nachvollziehbar und daher für die vorliegende Arbeit nicht von höherem Wert.

Rückblick und Ausblick

Der vorgegebene Rahmen der Maturaarbeit in Bezug auf Umfang, Nachvollziehbarkeit und Erstellungsdauer schränkte die verschiedenen Ideen zu dieser Arbeit markant ein. Ursprünglich war angedacht, eine deutlich umfangreichere Problemstellung zu bearbeiten. Aus diesem Grund wurde das eigentliche Potenzial von TensorFlow und Keras bei Weitem nicht ausgeschöpft. Dadurch könnte ein falscher Eindruck vom eigentlichen Potenzial dieser Tools entstehen. Wichtig ist daher zu erkennen, dass TensorFlow und Keras zu weit mehr im Stande sind, als das dargelegte Anwendungsbeispiel vermuten lässt.

Daher soll an dieser Stelle exemplarisch aufgezeigt werden, welche weiteren Anwendungen in einem grösseren Rahmen und unter Verwendung von TensorFlow und Keras ebenfalls bearbeitet werden können.

Variational Autoencoder



Wie im Vorwort erwähnt, bestand der ursprüngliche Plan dieser Arbeit darin, einen **Variational Autoencoder** (VAE) zu implementieren. Dieser sollte genutzt werden, um künstlich generierte Bilder von menschlichen Gesichtern zu erzeugen. Dies ist möglich, da es sich bei einem VAE um ein sogenanntes **Generatives Modell** handelt.

Generative Modelle sind in der Lage, anhand von einem existierenden Datensatz neue Dateneinträge zu generieren. Diese neuen Daten weisen einen ähnlichen Charakter auf wie der ursprüngliche Datensatz. Diese sind jedoch nicht in ihm enthalten.

Ein VAE erweitert das Grundkonzept eines normalen Autoencoders. Beim VAE ist das Encoding (Repräsentation) eine stochastische Verteilung, welche den entwickelten Code darstellt. Auf diese Weise wird bezweckt, dass Codes, welche im Repräsentationsraum nahe beieinander liegen, auch ähnliche Decodings (Rekonstruktionen) erzeugen. Dadurch wird der Repräsentationsraum kontinuierlich (siehe Abb. (6.8)). So können neue zufällige Codes rekonstruiert werden, welche sinnvolle Decodings erzeugen. Auf diese Weise agiert der Decoder als generatives Modell. Wenn beispielsweise ein VAE mit menschlichen Gesichtern trainiert würde, könnten neue zufällige Codes rekonstruiert werden, welche dann neue, noch nie gesehene Gesichter produzieren würde.

Eine verständliche und gute Erklärung bietet der Youtube-Kanal “Arxiv-Insights” in seinem Video “Variational Autoencoder” (siehe QR-Code).

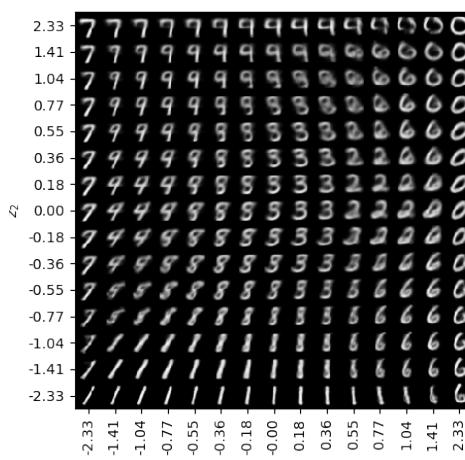


Abbildung 6.8: Decodingraum eines VAE trainiert auf MNIST: Ähnliche Ziffern sind nahe beieinander (31)

Generative Adversial Networks

Ein weiteres Generatives Modell sind sogenannte **Generative Adversial Networks** (GANs). Sie sind der anerkannte Status Quo, was generative KNNs anbelangt.

Das Grundkonzept ihrer Funktionsweise lautet wie folgt: Ein GAN besteht aus zwei KNNs, welche miteinander konkurrieren. Das eine Modell, der **Diskriminator**, kämpft gegen das andere Modell, den **Generator**. Die Aufgabe des Generator ist es, Daten zu generieren, welche möglichst stark dem Trainingsdatensatz ähneln. Der Diskriminator bekommt entweder ein Sample aus dem Trainingsdatensatz vorgelegt oder ein generiertes. Er muss dann entscheiden, ob es echt oder generiert ist. Beide Netzwerke lernen so, immer besser in ihrer Tätigkeit zu werden und betreiben gewissermassen einen Wettbewerb.

NVIDIA hat 2019 ein Paper veröffentlicht, in welchem sie ein GAN implementiert haben. Dieses Modell generierte Bilder von menschlichen Gesichtern. Die Ergebnisse sind von Auge nicht von echten Fotos zu unterscheiden. In Abbildung (6.9) ist ein Auszug an generierten Bildern dargestellt.

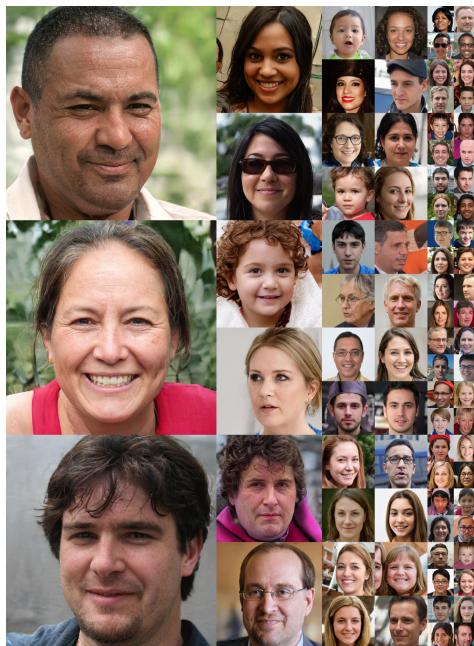


Abbildung 6.9: generierte Gesichter von NVIDIAs GAN (33)

Quellen: (33)

Frameworks

TensorFlow 2.0

TensorFlow befindet sich momentan in einer grossen Überarbeitungsphase. Es wächst gewissermassen aus seinen Kinderschuhen heraus. Es handelt sich dabei um das TensorFlow 2.0 Update. Bereits zum Zeitpunkt des Verfassens dieser Arbeit existiert eine Entwicklungsversion von TF 2.0, jedoch läuft diese noch nicht völlig stabil. Sobald TF 2.0 offiziell freigegeben wird, ist zu empfehlen, davon Gebrauch zu machen. In dieser Überarbeitung werden viele Relikte aus den ursprünglichen TensorFlow Versionen entfernt, welche für Verwirrung sorgen könnten. Somit entsteht ein in sich stimmiges Framework, welches einen höheren Komfort aufweisen soll.

Quelle: (34)

Alternativen

Es gibt neben TensorFlow und Keras noch weitere Frameworks, welche sich für Maschinelles Lernen eignen. Die meisten davon sind ebenfalls hauptsächlich auf Deep Learning ausgelegt. Sie machen auch Gebrauch von der cuDNN-Bibliothek von NVIDIA (erwähnt in Anhang (C)). Somit weisen sie ähnlich performante Implementierungen wie TF auf. Vorteile können in der Intuition und Einfachheit der Bedienung liegen. Viele Privatpersonen

bevorzugen aus diesem Grund **PyTorch** gegenüber TF. Weitere erwähnenswerte Frameworks sind: Caffe, Scikit-Learn, Sonnet.

Quelle: (2)

Schlusswort

Rückblickend war es für mich weitestgehend ein Vergnügen, eine Maturaarbeit zum Thema Maschinelles Lernen schreiben zu dürfen. Die Theorieabschnitte zu verfassen, bereitete mir wirklich viel Spass. Am Ende ein funktionierendes Programm erstellt zu haben, war ebenfalls ein Erfolgserlebnis für mich. Inhaltlich war es aber auch eine Herausforderung, die mich bis zum Schluss sehr umfassend beschäftigt hat, obwohl ich schon zu Beginn viele Stunden in diese Arbeit investiert habe. Durch mein privates Interesse am Programmieren und meine Praktika an der Universität Basel sowie bei Adobe konnte ich mit guten Grundlagen in das Thema einsteigen. Allerdings musste ich rasch feststellen, dass es sich um ein geradezu unerschöpfliches Thema handelt. Vor allem die mathematische Fundierung gestaltete sich aufwändiger als ich erwartet habe.

Reizvoll war ebenfalls, sich detailliert mit der aktuellen Fachliteratur auseinanderzusetzen und erste Eindrücke vom wissenschaftlichen Arbeiten zu erhalten. Angesichts der Breite des Themas war die thematische Einschränkung ein sinnvoller Schritt. Ich merkte rasch, dass mein ursprünglicher Anspruch, Bilder von menschlichen Gesichtern zu generieren, aufgrund des mathematischen Umfangs den Rahmen einer Maturaarbeit gesprengt hätte.

Die Faszination für das Thema und die zahlreichen Anwendungsmöglichkeiten geben mir die Gewissheit, dass ich auch inskünftig diesem Fachgebiet verbunden bleiben möchte. So bildet meine Maturaarbeit einen ersten Mosaikstein, auf den ich gerne zurückschauen werde.

Anhang A

Gradientenverfahren

Das Gradientenverfahren ist ein numerisches Verfahren, um Funktionen $f(x_1, x_2, \dots, x_n)$ vom Typ $\mathbb{R}^n \rightarrow \mathbb{R}$ (wie z. B. die Fehlerfunktion) zu minimieren. Dies geschieht, indem ein Startpunkt (Ortsvektor) \mathbf{p}_0 gewählt wird, dessen Komponenten den Input von $f(\mathbf{p}_0)$ darstellen. Nun werden iterativ neue Punkte \mathbf{p}_t gesucht, welche immer näher beim lokalen Minimum liegen, also Punkte, die den Funktionswert $f(\mathbf{p}_t)$ immer kleiner werden lassen. Dies wird durchgeführt, bis der Punkt genügend nahe beim lokalen Minimum ist.

Dafür muss ein Vektor \mathbf{b}_t bestimmt werden, welcher auf den Punkt \mathbf{p}_t addiert einen neuen Punkt \mathbf{p}_{t+1} bildet, bei dem der Funktionswert $f(\mathbf{p}_{t+1})$ kleiner ist als der von $f(\mathbf{p}_t)$. Dies geschieht am effizientesten, wenn \mathbf{b}_t in die Richtung der stärksten Funktionswertabnahme zeigt.

Hierzu wird der sogenannten **Gradient** ∇ verwendet, für den wiederum partielle Ableitungen benötigt werden.

Partielle Ableitungen

Partielle Ableitungen sind eine Erweiterung der “normalen” Ableitungen auf multidimensionale Funktionen $f(\mathbf{x}) = f(x_1, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}$. Man leitet dabei nur nach einem Argument x_i ab und betrachtet die restlichen Argumente als Konstanten. Es gelten die gleichen Ableitungsregeln wie bei der nicht-partiellen Ableitung. Die partielle Ableitung einer Funktion $f(x_1, \dots, x_n)$ bezüglich einer Variable x_i in einem Punkt $\mathbf{a} = (a_1 \ \dots \ a_n)^\top$ ist analog zur normalen Ableitung folgendermassen definiert:

$$\frac{\partial f}{\partial x_i}(\mathbf{a}) := \lim_{h \rightarrow 0} \frac{f(a_1, \dots, a_i + h, \dots, a_n) - f(a_1, \dots, a_i, \dots, a_n)}{h}$$

Geometrisch ist dies die Steigung der Tangente an die Kurve der Funktion f im Punkt \mathbf{a} . Die Tangente liegt in der Richtung der Achse des Parameters x_i , nach welchem abgeleitet wurde.

Gradient

Der Gradient ∇ ist ein Differentialoperator, der auf eine skalare Funktion $f(\mathbf{x}) = f(x_1, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}$ angewendet, das Skalarfeld auf das Gradientenfeld (Vektorfeld) abbildet. Um das Gradientenfeld ∇f der Funktion f zu bilden, fasst man alle partiellen Ableitungen der Funktion f , nach jedem Argument x_i , in einem Vektor zusammen. Um nun einen partikulären Vektor zu bestimmen, berechnet man den Gradienten in einem spezifischen Punkt $\mathbf{p} = (p_1 \ \dots \ p_n)^\top$:

$$\nabla_{\mathbf{x}} f(\mathbf{p}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{p}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{p}) \end{pmatrix}$$

Geometrisch ist der Gradient $\nabla f(\mathbf{p})$ einer Funktion f in einem Punkt \mathbf{p} der Vektor, welcher in die Richtung des steilsten Anstiegs von f zeigt. Sein Betrag gibt die Stärke des Anstiegs an.

Beispiel für partielle Ableitungen

Als Beispiel wird die Funktion $f(x, y) = x^2 + y^2 - 2$ betrachtet, welche von den Variablen x und y abhängt. Um die partielle Ableitung bezüglich x zu bestimmen, muss y als konstant betrachtet werden. Beispielsweise kann man $y = 0$ wählen, damit die Funktion $g(x) = f(x, 0) = x^2 - 2$ nur noch von der Variable x abhängt. Leitet man nun nach x ab, erhält man: $\frac{dg(x)}{dx} = g'(x) = 2x$. Dies entspricht der partiellen Ableitung von f nach x :

$$\frac{\partial f(x, y)}{\partial x} = 2x$$

Die partielle Ableitung von f nach y lautet entsprechend:

$$\frac{\partial f(x, y)}{\partial y} = 2y$$

Gradientenbeispiel

Als Beispiel für den Gradienten wird die Funktion $f(x, y) = x \cdot \exp(-x^2 - y^2)$ betrachtet. Werden die partiellen Ableitungen nach den Argumenten x und y gebildet, so erhält man (Kettenregel und Produktregel):

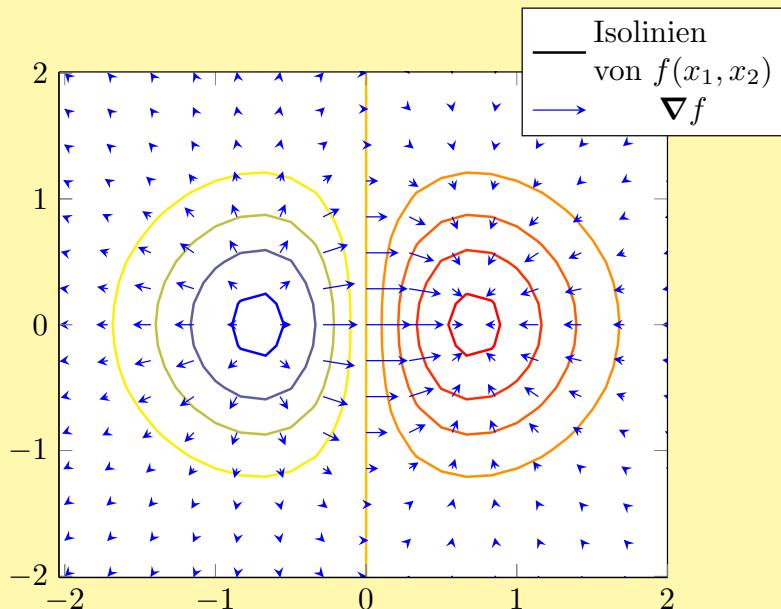
$$\frac{\partial f}{\partial x} = \exp(-x^2 - y^2) \cdot (1 - 2x^2)$$

$$\frac{\partial f}{\partial y} = \exp(-x^2 - y^2) \cdot (-2xy)$$

Diese beiden partiellen Ableitungen werden zum Gradient ∇f zusammengefasst:

$$\nabla f(x, y) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} \exp(-x^2 - y^2) \cdot (1 - 2x^2) \\ \exp(-x^2 - y^2) \cdot (-2xy) \end{pmatrix}$$

In der unterstehenden Abbildung ist das Konturdiagramm der Funktion f zusammen mit dem Gradientenfeld (blaue Vektoren) dargestellt.



Da der Gradient also in die Richtung des steilsten Anstiegs zeigt, sollte der Vektor \mathbf{b}_t gerade in die entgegengesetzte Richtung zeigen. Demnach sollte er in die Richtung des negierten Gradienten der Funktion f im Punkt \mathbf{p}_t weisen. Jetzt kann das iterative Annähern an das lokale Minimum folgendermassen beschrieben werden:

$$\mathbf{p}_{t+1} = \mathbf{p}_t - \alpha \cdot \nabla f(\mathbf{p}_t) \quad (\text{A.1})$$

Dabei stellt α einen Proportionalitätsfaktor dar, welcher die Schrittgrösse beim Abstieg bestimmt.

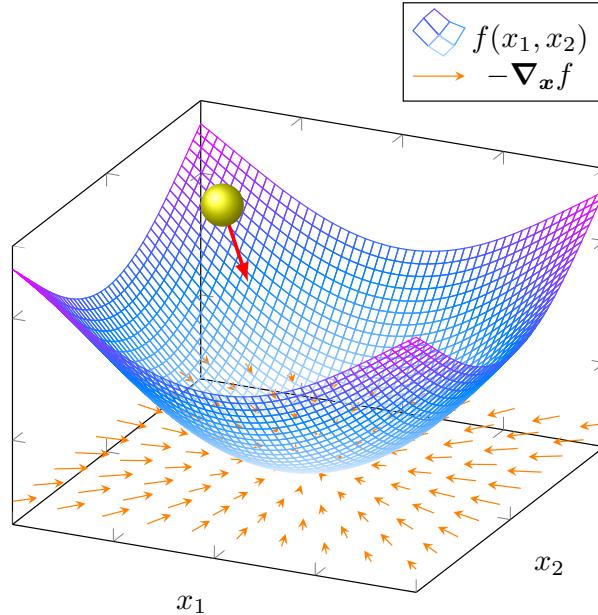


Abbildung A.1: Visualisierung des Gradientenabstiegs: Ein Ball rollt das Gradientenfeld hinab in das lokale Minimum.

Während des Gradientenverfahrens konvergiert der Punkt \mathbf{p}_t zu einem beliebigen *lokalen* Minimum, abhängig davon, wie der Startpunkt \mathbf{p}_0 gewählt wurde.

Quellen: (2) (3)

Anhang B

Herleitung der Rückwärtspropagierung für KNNs

Da ein KNN, wie der Name schon andeutet, vernetzt ist, können die partiellen Ableitungen einer Schicht anhand seiner Nachbarschichten berechnet werden. Dies ist auch der namensgebende Grundgedanke der Rückwärtspropagierung: Man beginnt damit, die partiellen Ableitung der letzten Schicht zu bestimmen und berechnet dann retrograd Schicht für Schicht die vorherigen partiellen Ableitungen bis zur Inputschicht.

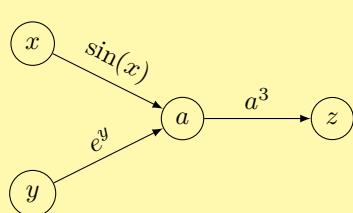
Dies geschieht unter Anwendung der Kettenregel der Ableitungen. Es ist sinnvoll, für das Aufstellen dieser Gleichungen das Netzwerk als **Computational Graph** zu betrachten.

Computational Graph

Ein Computational Graph ist die Darstellung einer Verkettung von Funktionen als Netzwerk von Operationen. Die Knoten im Graph stellen Variablen dar und die Pfade, welche die Knoten verbinden, sind die Funktionen, welche die Variablen aufeinander abbilden. Die Funktion wird auf die Variable angewendet, von der der Pfad ausgeht. Der Knoten, in welchem der Pfad endet, nimmt dann den Funktionswert an. Falls mehrere Pfade in einem einzigen Knoten enden, werden die einzelnen Werte der Pfade zusammenaddiert, um die Variable zu bilden. In diesem Graphen sind die Abhängigkeiten der Variablen voneinander gut ersichtlich. Auf dieser Basis können mithilfe der Kettenregel die Ableitungen unkompliziert bestimmt werden.

Beispiel eines Computational Graph

Untenstehend ist ein Beispiel eines Computational Graph zusammen mit der Herleitung der partiellen Ableitungen dargestellt.



$$\begin{aligned} a(x, y) &= \sin(x) + e^y \\ z(a(x, y)) &= a^3(x, y) = (\sin(x) + e^y)^3 \\ \frac{\partial z}{\partial x} &= \frac{\partial a}{\partial x} \cdot \frac{\partial z}{\partial a} = \cos(x) \cdot 3a^2 \\ \frac{\partial z}{\partial y} &= \frac{\partial a}{\partial y} \cdot \frac{\partial z}{\partial a} = e^y \cdot 3a^2 \end{aligned}$$

Der erste Schritt der Rückwärtspropagierung besteht darin, dass die partiellen Ableitungen $\frac{\partial C}{\partial z_j^l}$ der Kostenfunktion C bezüglich den gewichteten Summen z_j^l aller Schichten berechnet werden müssen. Daraus lassen sich dann später die partiellen Ableitungen bezüglich den Gewichten $\frac{\partial C}{\partial w_{j,k}^l}$ und bezüglich den Neigungen $\frac{\partial C}{\partial b_j^l}$ ermitteln.

Zur Übersichtlichkeit definiert man einen **Fehler** δ_j^l für jedes j -te Neuron in jeder l -ten Schicht, welcher die partielle Ableitung bezüglich der gewichteten Summe dieses Neurons darstellt (siehe Gl. (RP0)). Ebenfalls definiert man analog einen Fehlervektor $\boldsymbol{\delta}^l$, welcher alle Fehler δ_j^l einer Schicht l zusammenfasst (siehe Gl. (RP0a)). Nun heisst es, diesen Wert für jedes Neuron jeder Schicht zu berechnen.

$$\delta_j^l := \frac{\partial C}{\partial z_j^l} \quad (\text{RP0})$$

$$\boldsymbol{\delta}^l := \left(\frac{\partial C}{\partial z_1^l} \quad \frac{\partial C}{\partial z_2^l} \quad \cdots \quad \frac{\partial C}{\partial z_{|l|}^l} \right)^\top \quad (\text{RP0a})$$

Da die Kostenfunktion unmittelbar auf die letzte Schicht L angewendet wird, beginnt dort auch die Berechnung des Fehlers $\boldsymbol{\delta}^L$. Nun wird ein Computational Graph aufgestellt, um die partiellen Ableitungen zu bestimmen.

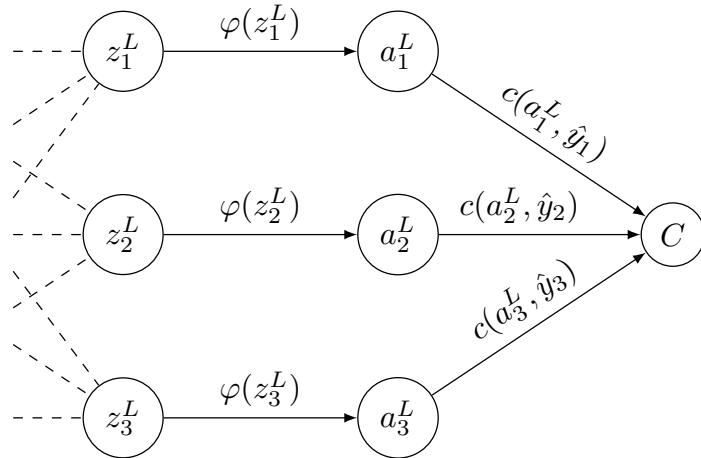


Abbildung B.1: Computational Graph zur Berechnung von $\boldsymbol{\delta}^L$

Aus dem Computational Graph der Abbildung (B.1) kann entnommen werden, dass die Kosten C eine Funktion in Abhängigkeit von den letzten Aktivierungen a_j^L ist. Diese ist wiederum eine Funktion in Abhängigkeit von der jeweiligen gewichteten Summe z_j^L . Somit kann mithilfe der Kettenregel die Beziehung (B.1) aufgestellt werden.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \quad (\text{B.1})$$

Da a_j^L durch die Anwendung der Aktivierungsfunktion φ auf z_j^L gebildet wird, ist $\frac{\partial a_j^L}{\partial z_j^L}$ die Ableitung der Aktivierungsfunktion $\varphi'(z_j^L)$. Auf diese Weise lässt sich die erste (RP1) von vier wichtigen Gleichungen für die Rückwärtspropagierung herleiten.

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \varphi'(z_j^L) \quad (\text{RP1})$$

Um diese Ausdrücke wieder in Matrixschreibweise zu realisieren, welche die ganze Schicht L zusammenfasst, muss eine neue Operation eingeführt werden: das Hadamard-Produkt.

Hadamard-Produkt

Das Hadamard-Produkt (auch elementweises Produkt) ist eine spezielle Multiplikation zweier gleichgrosser Matrizen $\mathbf{A} \in \mathbb{R}^{m \times n}$ und $\mathbf{B} \in \mathbb{R}^{m \times n}$. Die resultierende Matrix ergibt sich aus der elementweisen Multiplikation der Ausgangsmatrizen.

$$\mathbf{A} \odot \mathbf{B} = \begin{pmatrix} A_{1,1}B_{1,1} & \cdots & A_{1,n}B_{1,n} \\ \vdots & \ddots & \vdots \\ A_{m,1}B_{m,1} & \cdots & A_{m,n}B_{m,n} \end{pmatrix} \in \mathbb{R}^{m \times n} \quad \mathbf{v} \odot \mathbf{w} = \begin{pmatrix} v_1w_1 \\ \vdots \\ v_nw_n \end{pmatrix}$$

Mit φ' als die vektorisierte Ableitung der Aktivierungsfunktion kann der Fehlervektor der letzten Schicht nach Gleichung (B.2) berechnet werden.

$$\delta^L = \left(\frac{\partial C}{\partial a_1^L} \quad \frac{\partial C}{\partial a_2^L} \quad \cdots \quad \frac{\partial C}{\partial a_{|L|}^L} \right)^T \odot \varphi'(\mathbf{z}^L) \quad (\text{B.2})$$

Dabei ist der erste Operand des Hadamard-Produkts nichts anderes als der Gradient $\nabla_{\mathbf{a}^L} C$ der Kostenfunktion C bezüglich dem Aktivierungsvektor \mathbf{a}^L der letzten Schicht. Dieser Gradient kann ermittelt werden, indem die vektorisierte Ableitungsfunktion für die gewählte Kostenfunktion gebildet wird. Würde $C_{\text{MSE}} = \frac{1}{2|L|}(\hat{\mathbf{y}} - \mathbf{a}^L)^2$ als Kostenfunktion gewählt werden, ergäbe sich $\nabla_{\mathbf{a}^L} C = (\mathbf{a}^L - \hat{\mathbf{y}}) \cdot \frac{1}{|L|}$.

Daraus folgt die kompakte Matrix-Version (RP1a) der Gleichung (RP1), welche den Fehlervektor für die letzte Schicht berechnet.

$$\delta^L = \nabla_{\mathbf{a}^L} C \odot \varphi'(\mathbf{z}^L) \quad (\text{RP1a})$$

Nun soll eine rekursive Berechnungsmethode des Fehlers δ_j^{l-1} der vorherigen Schicht anhand des Fehlers δ_j^l der jetzigen Schicht erarbeitet werden. Zu diesem Zweck ist wiederum ein Computational Graph aufzustellen (siehe Abb. (B.2)).

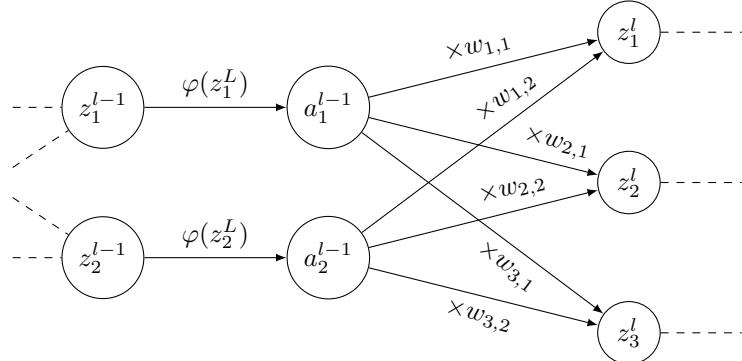


Abbildung B.2: Computational Graph zur Berechnung von δ_j^{l-1}

Es gilt erneut die Gleichung (B.1) für die Berechnung des Fehlers δ_j^{l-1} .

$$\delta_j^{l-1} = \frac{\partial C}{\partial z_j^{l-1}} = \frac{\partial a_j^{l-1}}{\partial z_j^{l-1}} \cdot \frac{\partial C}{\partial a_j^{l-1}} \quad ((\text{B.1}))$$

Der erste Faktor entspricht der Ableitung $\varphi'(z_j^{l-1})$ der Aktivierungsfunktion. Beim Übergang einer Schicht $(l-1)$ zur Schicht l beeinflusst eine Aktivierung a_j^{l-1} alle gewichteten Summen z_k^l . Mit der Kettenregel folgt daher, dass die partielle Ableitung $\frac{\partial C}{\partial a_j^{l-1}}$ die Summe aller $\frac{\partial C}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial a_j^{l-1}}$ sein muss. Daraus ergibt sich Gleichung (B.3).

$$\delta_j^{l-1} = \varphi'(z_j^{l-1}) \cdot \sum_{k=1}^{|l|} \left(\frac{\partial C}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial a_j^{l-1}} \right) \quad (\text{B.3})$$

Um die gewichtete Summe z_k^l zu bilden, wird Aktivierung a_j^{l-1} der vorherigen Schicht mit den entsprechenden Gewichten $w_{k,j}^{l-1}$ multipliziert. Dadurch entspricht diese partielle Ableitung $\frac{\partial z_k^l}{\partial a_j^{l-1}}$ gerade dem Gewicht selbst.

Des Weiteren ist $\frac{\partial C}{\partial z_k^l}$ per Definition der Fehler δ_k^l . Mit dieser Erkenntnis lässt sich die zweite essenzielle Gleichung (RP2) für die Rückwärtspropagierung aufstellen.

$$\delta_j^{l-1} = \varphi'(z_j^{l-1}) \cdot \sum_{k=1}^{|l|} (\delta_k^l \cdot w_{k,j}^{l-1}) \quad (\text{RP2})$$

Auch diese Gleichung soll in der Matrixschreibweise dargestellt werden. Zu diesem Zweck wird mit der Erweiterungen auf alle gewichteten Summen begonnen:

$$\boldsymbol{\delta}^{l-1} = \varphi'[\mathbf{z}^{l-1}] \odot \left(\sum_{k=1}^{|l|} w_{k,1}^{l-1} \cdot \delta_k^l \quad \dots \quad \sum_{k=1}^{|l|} w_{k,|l-1|}^{l-1} \cdot \delta_k^l \right)^T$$

Der zweite Operand des Hadamard-Produkts ist hierbei gerade das Produkt der Matrixmultiplikation zwischen der transponierten Gewichtsmatrix $(\mathbf{W}^{l-1})^T$ der Schicht $(l-1)$ und dem Fehlervektor $\boldsymbol{\delta}^l$ der Schicht l .

$$\begin{aligned} \left(\sum_{k=1}^{|l|} w_{k,1}^{l-1} \cdot \delta_k^l \quad \dots \quad \sum_{k=1}^{|l|} w_{k,|l-1|}^{l-1} \cdot \delta_k^l \right)^T &= \begin{pmatrix} w_{1,1}^{l-1} & w_{2,1}^{l-1} & \dots & w_{|l|,1}^{l-1} \\ w_{1,2}^{l-1} & w_{2,2}^{l-1} & \dots & w_{|l|,2}^{l-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,|l-1|}^{l-1} & w_{2,|l-1|}^{l-1} & \dots & w_{|l|,|l-1|}^{l-1} \end{pmatrix} \begin{pmatrix} \delta_1^l & \dots & \delta_{|l|}^l \end{pmatrix}^T \\ &= \begin{pmatrix} w_{1,1}^{l-1} & w_{1,2}^{l-1} & \dots & w_{1,|l-1|}^{l-1} \\ w_{2,1}^{l-1} & w_{2,2}^{l-1} & \dots & w_{2,|l-1|}^{l-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{|l|,1}^{l-1} & w_{|l|,2}^{l-1} & \dots & w_{|l|,|l-1|}^{l-1} \end{pmatrix}^T \boldsymbol{\delta}^l = (\mathbf{W}^{l-1})^T \boldsymbol{\delta}^l \end{aligned}$$

Damit wurde die rekursive Fehlerdefinition in Matrixschreibweise ausgedrückt. Und wir erhalten die kompakte Version (RP2a) der zweiten wichtigen Formel (RP2).

$$\boldsymbol{\delta}^{l-1} = ((\mathbf{W}^{l-1})^T \boldsymbol{\delta}^l) \odot \varphi'[\mathbf{z}^{l-1}] \quad (\text{RP2a})$$

In einem letzten Schritt sind nun noch die Formeln herzuleiten, mit welchen anhand des Fehlers δ_j^l die partiellen Ableitungen der Gewichte und der Neigungen berechnet werden können.

Eine Neigung b_j^l ist Funktionsbestandteil der entsprechenden gewichteten Summe z_j^{l+1} . Somit gilt für die Neigung Formel (B.4).

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial b_k^l} \quad (\text{B.4})$$

Der erste Term ist hierbei per Definition der Fehler δ_j^{l+1} und der zweite Term lässt sich auf 1 kürzen, da die Summe z_j^{l+1} nur aus b_k^l besteht und aus Summanden, welche für die partielle Ableitung als konstant gelten. Somit entspricht die Ableitung der Neigung gerade dem Fehler, womit die dritte (RP3) von vier essenziellen Gleichungen ermittelt wurde.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^{l+1} \quad (\text{RP3})$$

Somit entspricht der Kostengradient bezüglich der Neigung dem Fehlervektor.

$$\nabla_{b^l} C = \boldsymbol{\delta}^{l+1} \quad (\text{RP3a})$$

Ein Gewicht $w_{j,k}^l$ ist ebenfalls ein Funktionsbestandteil der assoziierten gewichteten Summe z_j^{l+1} . Dadurch gilt für die partiellen Ableitungen der Kosten nach dem Gewicht die Gleichung (B.5).

$$\frac{\partial C}{\partial w_{j,k}^l} = \frac{\partial C}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial w_{j,k}^l} \quad (\text{B.5})$$

Dabei entspricht der erste Teil der Gleichung wieder dem Fehler δ_j^{l+1} . Die zweite partielle Ableitung ist gerade die Aktivierung a_k^l , da sich die gewichtete Summe aus der Multiplikation des Gewichtes mit der Aktivierung ergibt. Auf diese Weise entsteht die letzte der vier essenziellen Gleichungen (RP4).

$$\frac{\partial C}{\partial w_{j,k}^l} = \delta_j^{l+1} \cdot a_k^l \quad (\text{RP4})$$

Die Matrix-Version lässt sich als Gleichung (RP4a) ausdrücken.

$$\nabla_{W^l} C = \delta^{l+1}(\mathbf{a}^l)^\top \quad (\text{RP4a})$$

Zusammenfassung Rückwärtspropagierung

0. Vorwärtspropagierung durchführen und dabei alle Zwischenwerte beibehalten.
1. Berechnung des Fehlers δ^L der letzten Schicht, anhand der Formel:

$$\delta^L = \nabla_{a^L} C \odot \varphi'(z^L) \quad (\text{RP1a})$$

2. Rekursive Berechnung des Fehlers δ^{l-1} der jeweils vorherigen Schicht, anhand der Formel:

$$\delta^{l-1} = ((W^{l-1})^\top \delta^l) \odot \varphi'[z^{l-1}] \quad (\text{RP2a})$$

3. Berechnung des Kostengradienten bezüglich der Neigungen, anhand der Formel:

$$\nabla_{b^l} C = \delta^{l+1} \quad (\text{RP3a})$$

4. Berechnung des Kostengradienten bezüglich den Gewichten, anhand der Formel:

$$\nabla_{W^l} C = \delta^{l+1}(\mathbf{a}^l)^\top \quad (\text{RP4a})$$

5. Gewichte und Neigung mit SGD aktualisieren

(3) (35) (36)

Anhang C

Performance von TensorFlow

Devices

Die Kernstücke des TensorFlow Backends sind die verschiedenen **Devices**, auf welchen die Berechnungen ausgeführt werden. Ein Device ist jegliche Art von Computerprozessor, auf welchem die Computational Graphs ausführt werden können. Zu diesen Prozessoren gehören die CPU (Hauptprozessor) und die GPU (Grafikprozessor), falls die CUDA-Version von TF installiert wurde.

Das Backend analysiert die verfügbaren Devices und bewertete sie bezüglich ihren Fähigkeiten. Anhand dieser Bewertung wird dann entschieden, auf welchem Device die Berechnungen ausgeführt werden¹. Insofern GPUs zur Verfügung stehen, wählt TF grundsätzlich immer diese, da sie Tensoroperationen deutlich schneller als CPUs verarbeiten können.

Performance und Hardwarebeschleunigung

Nun möchten wir die Gründe für die beachtliche Performance von TF erschliessen. Der erste wichtige Faktor dabei ist, dass das Backend grösstenteils in C++ geschrieben ist. C++ ist eine Programmiersprache die, sehr nahe am Maschinen-Code (engl.: low-level) ist. Somit ist der Code sehr hardwarenah und kann so schneller ausgeführt werden. Der Nachteil besteht darin, dass die Programmiersprache ziemlich aufwendig ist. Das bedeutet, man braucht viel Programmcode, um eine relativ einfache Idee auszudrücken. Deshalb wurde der Client von TF in Python geschrieben, was Abhilfe verschafft und so ein relativ einfaches Entwickeln ermöglicht.

Der andere wichtige Aspekt für die Performance ist die sogenannte **Hardwarebeschleunigung** (engl. hardware acceleration), von welcher TF Gebrauch macht. Hardwarebeschleunigung bezeichnet eine Sammlung an Methoden, bei welchen man spezialisierte Hardware verwendet, um rechenintensive Aufgaben schneller auszuführen. Die Architektur einer CPU ist zwar so ausgelegt, dass sie beliebige Aufgaben ausführen kann, jedoch meistens nicht besonders effizient. Deshalb existieren einerseits alternative externe Hardwarebausteine, wie die GPU, welche sich für Grafikberechnungen besonders eignen. Andererseits wurde auch die interne Architektur der CPU so angepasst, dass sie auch spezialisierte Aufgaben besser meistert.

Eine solche Architekturanpassung stellt die **Parallelisierung** von Operationen dar. Dabei spaltet man eine rechenintensive Aufgabe in viele Teilaufgaben, welche gleichzeitig in gleicher Weise ausgeführt werden. Voraussetzung dafür ist, dass die Teilaufgaben alle die gleichen Schritte umfassen und unabhängig voneinander bearbeitet werden können. Ein typisches Beispiel für die Parallelisierbarkeit von Operationen ist die Matrixmultiplikation. Möchte man eine Matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ mit einer zweiten Matrix $\mathbf{B} \in \mathbb{R}^{m \times p}$ multiplizieren, muss man praktisch m -mal den gleichen Schritt ausführen: Man berechnet jeweils das Skalarprodukt zwischen der i -ten Zeile von \mathbf{A} und der i -ten Spalte von \mathbf{B} . Alle diese Skalarprodukte haben keinen Einfluss aufeinander und können so unabhängig mit der gleichen Prozedur berechnet werden.

Allgemein eignen sich Tensoroperationen zur Parallelisierung und damit zur Hardwarebeschleunigung. Nun ist auch ersichtlich, weshalb TensorFlow als Hauptdatentyp Tensoren verwendet.

CPU-Beschleunigung

Die Hardwarebeschleunigung der CPU ist für TensorFlow nur von geringfügiger Relevanz, da TF sich vor allem für GPU-Hardwarebeschleunigung eignet und dafür optimiert ist. Außerdem ist es praktisch unmöglich, die gleiche Performance von TF mit einer CPU zu erreichen, welche mit einer GPU möglich ist.

¹Es ist möglich, die Graphen auf mehreren Devices gleichzeitig auszuführen und auf diese Weise die Modelle noch schneller zu trainieren. Dies ist jedoch erst für sehr aufwendige Projekte sinnvoll und erfordert ein fortgeschrittenes Verständnis.

Die Parallelisierung auf der CPU wird einerseits durch sogenanntes **SIMD** (“single instruction, multiple data”) bewerkstelligt. Dies stellt ein Prinzip dar, welches die gleiche Instruktion parallel auf mehrere Dateneinheiten ausführt. Andererseits werden auf der CPU die ganzen Berechnungen auf mehreren sogenannten **Threads** ausgeführt. Da fast alle heutigen CPUs mehrere Kerne besitzen, kann nicht nur auf einem einzigen, sondern gleich auf mehreren Kernen gerechnet werden.

GPU-Beschleunigung

Die Hardwarebeschleunigung für die Grafikkarte wird durch ein spezielles externes Framework erreicht. TensorFlow verwendet **cuDNN** (“NVIDIA CUDA deep learning neural network library”) eine von NVIDIA entwickelte Programmzbibliothek, um Deep Learning performant auf einer NVIDIA-GPU auszuführen. Praktisch alle professionellen Deep-Learning-Frameworks machen Gebrauch davon. Sie ist in der Programmiersprache **CUDA** geschrieben, welche ebenfalls von NVIDIA entwickelt wurde. Zudem ist sie eine zu C++ sehr ähnliche Programmiersprache, welche jedoch nicht auf einer CPU ausgeführt wird, sondern auf einer GPU.

Quelle: (37)

Anhang D

GitHub

Auf meinem GitHub-Account (LU15W1R7H) existiert ein Repository, welches verschiedene Dateien enthält, die für die vorliegende Arbeit relevant sind. Der mit `code` betitelte Ordner enthält sämtliche Programmcodes, welche in der Arbeit erwähnt wurden. Im Ordner `lateX` befinden sich alle L^AT_EX-Dateien, die zur Kompilierung der Arbeit selbst benötigt werden. Der untenstehende QR-Code ist ein direkter Link zum GitHub-Repository dieser Maturaarbeit.



Abbildung D.1: QR-Code zur URL:

<https://github.com/LU15W1R7H/matura>

Literatur

1. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, <http://www.deeplearningbook.org> (MIT Press, 2016).
2. A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow* (O'Reilly Media, 2017).
3. M. Nielsen, *Neural Networks and Deep Learning*, <http://neuralnetworksanddeeplearning.com>.
4. Wikipedia, *Überanpassung — Wikipedia, Die freie Enzyklopädie*, [Online; besucht am 31. August 2019], 2019, (<https://de.wikipedia.org/wiki/%C3%9Cberanpassung>).
5. Wikipedia, *Perzepron — Wikipedia, Die freie Enzyklopädie*, [Online; besucht am 28. April 2019], 2019, (<https://de.wikipedia.org/wiki/Perzepron>).
6. Wikipedia, *Linear separability — Wikipedia, The Free Encyclopedia*, [Online; besucht am 9. Juli 2019], 2019, (https://en.wikipedia.org/wiki/Linear_separability).
7. Wikipedia, *Künstliches Neuron — Wikipedia, Die freie Enzyklopädie*, [Online; besucht am 4. Mai 2019], 2019, (https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron).
8. Wikipedia, *Sigmoidfunktion — Wikipedia, Die freie Enzyklopädie*, [Online; besucht am 4. Mai 2019], 2019, (<https://de.wikipedia.org/wiki/Sigmoidfunktion>).
9. Wikipedia, *Künstliches Neuronales Netz — Wikipedia, Die freie Enzyklopädie*, [Online; besucht am 04. Mai 2019], 2019, (https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz).
10. Wikipedia, *Normal distribution — Wikipedia, The Free Encyclopedia*, [Online; besucht am 26. Juni 2019], 2019, (https://en.wikipedia.org/wiki/Normal_distribution).
11. Wikipedia, *Universal approximation theorem — Wikipedia, The Free Encyclopedia*, [Online; besucht am 22. August 2019], 2019, (https://en.wikipedia.org/wiki/Universal_approximation_theorem).
12. A. NG, *Convolutional Neural Networks (Course 4 of the Deep Learning Specialization)*, [Online; besucht am 17. Juni 2019], 2017, (<https://www.youtube.com/playlist?list=PLkDaE6sCZn6G129AoE31iwdVwSG-KnDzF>).
13. Wikipedia, *Convolutional neural network — Wikipedia, The Free Encyclopedia*, [Online; besucht am 25. Juni 2019], 2019, (https://en.wikipedia.org/wiki/Convolutional_neural_network).
14. Wikipedia, *Tensor — Wikipedia, Die freie Enzyklopädie*, [Online; besucht am 13. Juli 2019], 2019, (<https://de.wikipedia.org/wiki/Tensor>).
15. M. Richter, *Weisskopfadler*, [Online; besucht am 18. August 2019], 2019, (<https://pixabay.com/de/photos/wei%C3%9Fkopfseeadler-vogel-4370916/>).
16. Wikipedia, *Kernel (image processing) — Wikipedia, The Free Encyclopedia*, [Online; besucht am 25. Juni 2019], 2019, ([https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))).
17. *the GIMP Documentation: Convolution Matrix*, [Online; besucht am 22. September 2019], 2019, (<https://docs.gimp.org/2.8/en/plug-in-convmatrix.html>).
18. Wikipedia, *Convolution — Wikipedia, The Free Encyclopedia*, [Online; besucht am 25. Juni 2019], 2019, (<https://en.wikipedia.org/wiki/Convolution>).
19. E. Saalmann, *Einführung in Autoencoder und Convolutional Neural Networks* (Universität Leipzig, 2018).
20. T. Karras, S. Laine, T. Aila, *CoRR abs/1812.04948*, arXiv: 1812.04948, (<http://arxiv.org/abs/1812.04948>) (2018).
21. V. N. Marivate, F. V. Nelwamondo, T. Marwala, *CoRR abs/0709.2506*, arXiv: 0709.2506, (<http://arxiv.org/abs/0709.2506>) (2007).
22. CodeParade, *Computer Generates Human Faces*, [Online; besucht am 21. August 2019], Youtube (<https://www.youtube.com/watch?v=4VAkrUNLKSo>).

23. X. Mao, C. Shen, Y. Yang, *CoRR abs/1606.08921*, arXiv: 1606.08921, (<http://arxiv.org/abs/1606.08921>) (2016).
24. Wikipedia, *Additive white Gaussian noise — Wikipedia, The Free Encyclopedia*, [Online; besucht am 23. September 2019], 2019, (https://en.wikipedia.org/wiki/Additive_white_Gaussian_noise).
25. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015, (<https://www.tensorflow.org/>).
26. *TensorFlow Core r1.14 API-Docs*, [Online; besucht am 14. September 2019], 2019, (https://www.tensorflow.org/api_docs/python/tf).
27. *Keras Documentation*, [Online; besucht am 03. September 2019], (<https://keras.io/initializers/>).
28. J. Steppan, *File:MnistExamples.png — Wikipedia, The Free Encyclopedia*, [Online; besucht am 18. August 2019], 2017, (<https://en.wikipedia.org/wiki/File:MnistExamples.png>).
29. C. J. B. Yann LeCun Corinna Cortes, *The MNIST Database of handwritten digits*, [Online; besucht am 15. August 2019], (<http://yann.lecun.com/exdb/mnist/>).
30. F. Chollet, *Keras Autoencoder Tutorial*, [Online; besucht am 26. September 2019], 2016, (<https://blog.keras.io/building-autoencoders-in-keras.html>).
31. L. Tiao, *Implementing Variational Autoencoders in Keras: Beyond the Quickstart Tutorial*, [Online; besucht am 22. September 2019], 2017, (<http://louistiao.me/posts/implementing-variational-autoencoders-in-keras-beyond-the-quickstart-tutorial/>).
32. Wikipedia, *Autoencoder — Wikipedia, The Free Encyclopedia*, [Online; besucht am 21. September 2019], 2019, (<https://en.wikipedia.org/wiki/Autoencoder>).
33. T. Karras, S. Laine, T. Aila, *CoRR abs/1812.04948*, arXiv: 1812.04948, (<http://arxiv.org/abs/1812.04948>) (2018).
34. *Effective TensorFlow 2.0*, [Online; besucht am 23. September 2019], 2019, (https://www.tensorflow.org/beta/guide/effective_tf2).
35. R. Rojas, *Neural Networks — A Systematic Introduction* (Springer-Verlag, 1996).
36. R. Merz, *Wie funktioniert Deep Learning?*, 2018.
37. NVIDIA cuDNN, [Online; besucht am 23. September 2019], 2019, (<https://developer.nvidia.com/cudnn>).

Abbildungsverzeichnis

1.1	Visualisierung von Under- und Overfitting	6
2.1	Perzepron mit drei Inputs	8
2.2	zwei-dimensionale lineare Separierung	8
2.3	ein künstliches Neuron	9
2.4	Definition und Graph der Heaviside-Funktion Θ	9
2.5	Formel und Graph der ReLU-Funktion	10
2.6	Definition, Ableitung und Graph der Sigmoid-Funktion σ	10
2.7	Schichten eines KNNs	11
2.8	zum Verständnis der Nomenklatur der Neuronen	12
2.9	zum Verständnis der Gewichtebeschriftungen	12
2.10	Graph der Dichtefunktion $\phi(x \mu = 0, \sigma^2 = 1)$ mit ihren wichtigsten Eigenschaften	15
3.1	Schichtung eines CNNs	18
3.2	der Instagramfilter "Amaro" auf ein Beispielbild angewandt (15)	18
3.3	eine 2D-Filtermatrix mit rotem Zentralelement (\mathbf{F}) _C	18
3.4	Kantendetektionfilter angewandt auf Beispieldbild (15)	19
3.5	ein Filter und sein rezeptives Feld	20
3.6	Schema, wie ein Filter über ein Bild läuft	21
3.7	ein 3D-Filtertensor mit rotem Zentralelement (\mathbf{F}) _C	22
3.8	Padding $p = 1$ in blau	23
3.9	Abbildung zum Stride: Ausgangsposition in blau, nächste Position mit $s = 1$ in rot, nächste Position mit $s = 2$ in grün	23
4.1	Schichten eines kurzen Autoencoders	28
4.2	Schichten eines tieferen Autoencoders	28
4.3	Darstellung eines Convolutional Autoencoders	30
4.4	Visualisierung eines Convolutional-Denoising-Autoencoder	31
4.5	zwei-dimensionales Gauss'sches Rauschen mit Rauschfaktor $c = 1$	31
5.1	QR-Code zum GitHub-Repo	32
5.2	QR-Code zu den Docs	33
5.3	Beispielsgraph, welcher sich wie ein künstliches Neuron verhält	34
5.4	das TensorBoard-Webinterface	35
5.5	Trainingsvisualisierungen in TensorBoard	36
6.1	Schema des beschriebenen Modells (Tensoren sind nicht massstabsgetreu)	39
6.2	ein Auszug an MNIST-Bildern (invertierte Farben) (28)	40
6.3	QR-Code für Installationsanweisungen	40
6.4	Verrauschte Bilder neben den Originalbildern	43
6.5	Verrauschte Bilder, Rekonstruktionen und Originalbilder	47
6.6	Kosten bezüglich dem Testdatensatz	48
6.7	Sehr verrauschte Bilder, Rekonstruktionen und Originalbilder	49
6.8	Decodingraum eines VAE trainiert auf MNIST: Ähnliche Ziffern sind nahe beieinander (31)	51
6.9	generierte Gesichter von NVIDIA's GAN (33)	52
A.1	Visualisierung des Gradientenabstiegs: Ein Ball rollt das Gradientenfeld hinab in das lokale Minimum	57
B.1	Computational Graph zur Berechnung von δ^L	59

B.2 Computational Graph zur Berechnung von δ_j^{l-1}	60
D.1 QR-Code zur URL: https://github.com/LU15W1R7H/matura	65

Tabellenverzeichnis

6.1 Rangliste der besten Modelle mit ihren Eigenschaften	49
--	----

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig durchgeführt und keine anderen als die angegebene Quellen, Hilfsmittel und Hilfspersonen beigezogen habe. Alle Textstellen in der Arbeit, die wörtlich oder sinngemäß aus Quellen entnommen wurden, habe ich als solche gekennzeichnet.

Luis Wirth