Künstliche neuronale Netze
Deep Convolutional Neural Networks

Jens Ostertag

16. Juni 2022

Inhaltsverzeichnis

1 Computer lernen lassen

Künstliche Intelligenz, *Maschinelles Lernen* und *Neuronale Netze* - Alle diese Begriffe sind gewissermaßen miteinander verbunden, denn ihre Technik beruht darauf, dass Computer lernen können. Vor einem tieferen Einblick möchte ich diese Begriffe jedoch einordnen und differenzieren.

Bei einem *künstlichen neuronalen Netz* handelt es sich um die Simulation des menschlichen Gehirns. Neuronale Netze sind die Grundlage für alles, das Lernen kann oder Entscheidungen treffen soll.

Maschinelles Lernen beschreibt das Lernen eines solchen neuronales Netzes. Wie auch der Mensch, ist ein künstliches neuronales Netz in der Lage, sich unterschiedliche Fähigkeiten anzueignen, unabhängig von der Komplexität der jeweiligen Aufgabe. Dadurch wird beispielsweise ermöglicht, dass ein Computer Bilder erkennt und klassifiziert oder die Position bestimmter Objekte darin bestimmen kann.

Die *künstliche Intelligenz* ist die undefinierteste Form neuronaler Netze, was auch damit zusammenhängt, dass es keine sehr genaue Definition von Intelligenz gibt. Ist man intelligent, wenn man schnell rechnen kann? Unter dieser Voraussetzung wäre bereits ein simpler Taschenrechner von vor 40 Jahren äußerst intelligent. Oder bedeutet Intelligenz, dass man selbstständig denken und fühlen können muss, in dessen Zusammenhang der Begriff „Künstliche Intelligenz“ auch meistens verwendet wird? Aufgrund dieser Undefiniertheit wird im Folgenden, sofern möglich, auf diesen Begriff verzichtet.

Obwohl die Theorie schon etwas länger existiert, erfolgte ein großer Vorsprung in der Entwicklung des maschinellen Lernens erst in den letzten Jahren. Auch wenn es zuerst nicht so scheint, kommt heute jeder täglich damit in Berührung, beispielsweise

- im Verkehr, durch möglichst effizient gesteuerte Ampelphasen oder nahezu selbstfahrende Autos sowie Assistenzsysteme wie Spurhalteassistenten, Einparkhilfen oder Ähnliches,
- im Internet, wo jedem im Sekundentakt neue Inhalte basierend auf persönlichen Präferenzen vorgeschlagen werden,
- in der Medizin, wo Bilderkennung bei der Auswertung bildgebender Verfahren hilft,
- in der Industrie, wo Produktionsabläufe durch maschinelles Lernen optimiert wurden

und natürlich in vielen weiteren aufwendigeren Prozessen.

Im Folgenden soll es darum gehen, wie ein neuronales Netz funktioniert und wie man es mit maschinellem Lernen verwenden kann.

1.1 Aufbau neuronaler Netze [1]

Künstliche neuronale Netze sind sehr stark an natürliche neuronale Netze angelehnt (zum Beispiel an das menschliche Gehirn), weshalb es sinnvoll ist, zuerst diese zu betrachten.

Sogenannte *Neuronen* (Nervenzellen) sind durch *Synapsen* miteinander verbunden, welche für den Elektronenfluss zwischen jeweils zwei Neuronen mithilfe eines *Synapsengewichts* verantwortlich sind. Mit diesem lässt sich die Anfälligkeit des nachgehenden Neurons auf das Signal des vorhergehenden Neurons regulieren. [4]

Es ergibt sich somit das folgende Bild:

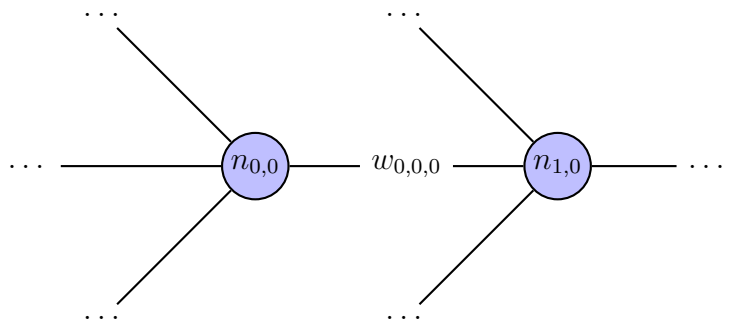


Abbildung 1: Synapse zwischen zwei Neuronen

Es sei jedoch angemerkt, dass ein Neuron beliebig viele vorausgehende oder nachgehende Neuronen haben kann, die in Abbildung 1 der Übersicht halber nur durch „$\cdots$“ gekennzeichnet sind. Als grobe Größenordnung sei die Anzahl der Neuronen eines ausgewachsenen, menschlichen Gehirns genannt, welche sich je nach Literatur auf 80 bis 100 Milliarden beläuft.

Eine wichtige Grundlage, um aus dem natürlichen neuronalen Netz ein Künstliches abzuleiten, ist das Betrachten von Ausgaben von Neuronen und von Synapsengewichten als Zahlen. Das ermöglicht die Anwendung von Formeln zur mathematischen Beschreibung.

Ebenso wichtig ist die Einteilung aller Neuronen des Netzes in unterschiedliche Schichten, die teilweise sogar unterschiedliche Aufgaben übernehmen

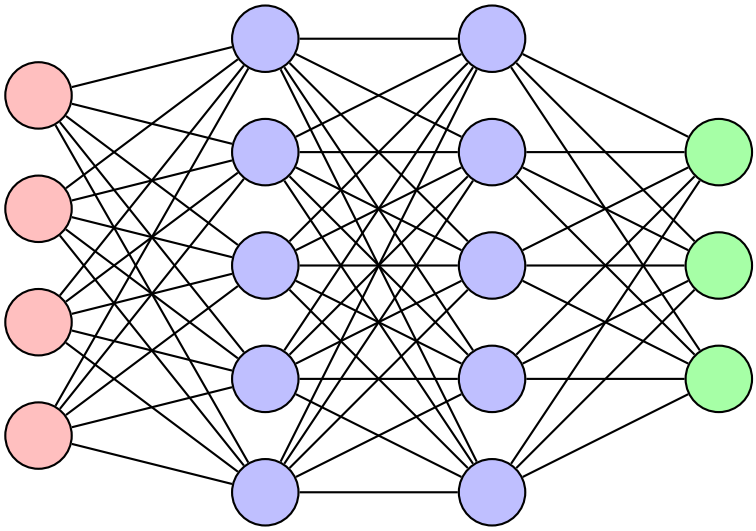


Abbildung 2: Aufbau eines einfachen neuronalen Netzes

müssen. Als Veranschaulichung dafür dient Abbildung 2, die den Aufbau eines einfachen neuronalen Netzes darstellt.

Die erste Schicht ist die sogenannte „Eingabeschicht“, dessen Neuronen (rot) vergleichbar sind mit Sinneszellen. Hier werden Daten gesammelt, mit denen eine Ausgabe generiert werden soll. Es folgen beliebig viele „versteckte Schichten“. Diese erfüllen den Zweck, dass das Netz auch kompliziertere Dinge lernen kann, da die Gesamtheit der Gewichte ohne versteckte Schicht nur eine linear separierbare Funktion darstellen könnte. Abschließend folgt eine „Ausgabeschicht“, welche beispielsweise Entscheidungen des Netzes ausgibt, welche dann in einem anderen Teil des Programms verarbeitet werden. In diesem Fall sind alle Neuronen einer Schicht mit allen Neuronen der nächsten Schicht verbunden. Wie jedoch in Kapitel 2 noch thematisiert wird, existieren auch weitere Strukturen für den Aufbau eines neuronalen Netzes.

Diese Grundlagen ermöglichen beispielsweise eine Berechnung der Ausgabe eines Neurons mit der Formel

$$o_{i,j} = \varphi\left(\sum_{k=0}^{|n_{i-1}|-1} o_{i-1,k} * w_{i-1,k,j}\right) \tag{1}$$

mit

- $n_{i,j}$: Neuron in der Schicht i an der Stelle j

- $o_{i,j}$: Ausgabe des Neurons $n_{i,j}$
- φ: Differenzierbare Aktivierungsfunktion
- $|n_i|$: Anzahl der Neuronen in der Schicht i
- $w_{i,k,j}$: Synapsengewicht zwischen den Neuronen $n_{i,k}$ und $n_{i+1,j}$

Wörtlich bedeutet das, dass zuerst die Netzeingabe eines Neurons durch Addieren aller Produkte einer Ausgabe eines vorherigen Neurones mit dem jeweiligen Gewicht zwischen den beiden Neuronen berechnet wird. Die Ausgabe des Neurons lässt sich nun mit der *Aktivierungsfunktion* in Abhängigkeit dieser Summe berechnen. Eine Aktivierungsfunktion wird benötigt, damit ein neuronales Netz auch nicht-lineare Eigenschaften erlernen kann.

1.2 Drei Arten des maschinellen Lernens [2]

Damit ein neuronales Netz lernt, müssen seine Gewichte angepasst werden. Dieser Vorgang wird auch Training genannt. Das kann auf drei unterschiedliche Arten erfolgen.

1.2.1 Überwachtes Lernen

Das überwachte Lernen ist die wohl am häufigsten verwendete Methode, ein neuronales Netz zu trainieren. Es zielt darauf ab, auch zu bislang unbekannten Eingaben eine passende Ausgabe zu generieren, meistens geht es darum, die Eingaben in Gruppen bestimmter Eigenschaften einzuteilen.

Während des Trainingsvorgangs wird mithilfe eines Trainingsdatensatzes (einige Eingaben mit zugehörigen erwarteten Ausgaben) angelernt, welche Muster in den Eingabedaten zu bestimmten Ausgaben gehören. Erhält das Netz nach abgeschlossenem Training eine Eingabe, ist es in der Lage, ein Muster darin zu erkennen und anhand dessen eine passende Ausgabe zu generieren.

In Abbildung 3 sind in einem Diagramm mehrere Punkte unterschiedlicher Farben eingetragen, die jeweils einem x- und y-Wert zugeordnet sind. Jeder Punkt entspricht dabei einer Eingabe, alle Eingaben derselben Farbe gehören der gleichen Klasse an. Während des Trainings lernt das Netz beispielsweise, dass eine Eingabe mit hohem x- und niedrigem y-Wert in die grüne Klasse einteilen soll.

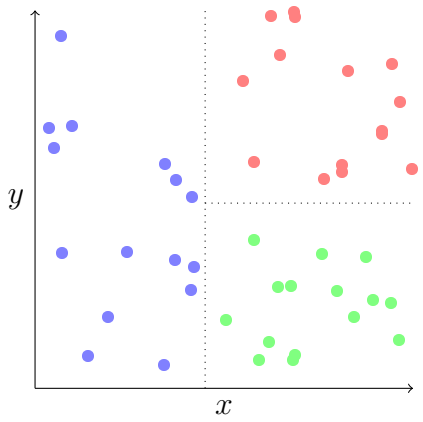


Abbildung 3: Klassifizierung von Daten

1.2.2 Unüberwachtes Lernen

Das Ziel des unüberwachten Lernens ist es, innerhalb einer Menge von Eingaben Gruppen anhand ähnlicher Strukturen zu finden. Dafür wird keine Bearbeitung oder Sortierung der Daten benötigt.

Daher ist diese Lernart besonders interessant für die Clusteranalyse, welche dasselbe Ziel anstrebt. Sie spielt besonders im Internet eine Rolle, wo Nutzer unterschiedlicher Zielgruppen unterschiedliche Inhalte, Produktvorschläge oder Werbungen angezeigt bekommen sollen.

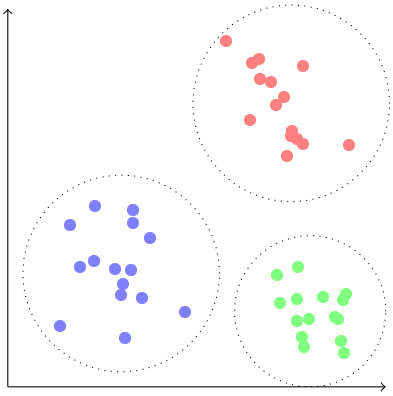


Abbildung 4: Gruppierung von Daten

In Abbildung 4 sind in einem Diagramm mehrere Punkte unterschiedlicher Farben eingetragen, die jeweils einem x- und y-Wert zugeordnet sind.

Jeder Punkt entspricht dabei einer Eingabe. Anhand der Darstellung ist ersichtlich, dass die x- und y-Werte von Eingaben gleicher Farbe ähnlich sind, im Vergleich zu den Werten anderer Farben. Diese Ähnlichkeiten gilt es während des Lernvorgangs zu erkennen, sodass auch neu hinzukommende Daten einer der Gruppen zugeordnet werden könnte.

1.2.3 Bestärktes Lernen

Bestärktes Lernen dient dazu, den Entscheidungsprozess eines neuronales Netzes bezüglich einer bestimmten Tätigkeit zu trainieren.

Dabei wird nach dem Belohnungs-Prinzip gearbeitet: Unterschiedliche Ausgaben erhalten verschiedene Bewertungen, eine Zahl, die einen Vergleich zwischen Gut und Schlecht ermöglichen soll. Eine „gute“ Ausgabe wird belohnt, was darin resultiert, dass die jeweilige Entscheidung verstärkt wird. Entschied sich das Netz falsch, wird es nicht belohnt (oder sogar bestraft) und die Entscheidung wird geschwächt. Während des Lernvorgangs wird stets versucht, die Bewertung zu maximieren, um immer die bestmögliche Ausgabe zu erreichen.

Die Bewertungen sind hierbei eine Art Feedback für das Netz, weshalb man auch von einer Art des überwachten Lernens sprechen kann.

1.3 Umgang mit neuronalen Netzen [2]

Vor der Implementation maschinellen Lernens in einem Programm muss ein neuronales Netz entwickelt werden, das die gewünschte Aufgabe zuverlässig erledigen kann. Für die einzelnen Lernarten existieren standardisierte Abläufe, die von Netz zu Netz nahezu gleich sind. Im Folgenden wird der des überwachten Lernens vorgestellt.

1.3.1 Vorbereiten eines Trainingsdatensatzes

Zuerst müssen Daten gesammelt und zu einem *Trainingsdatensatz* zusammengefügt werden. Bei allen Daten handelt es sich um mögliche Eingaben für das Netz. Sollen also beispielsweise Bilder von unterschiedlichen Objekten unterschieden werden, sind die Daten ebenfalls Bilder, auf denen die zu klassifizierenden Objekte enthalten sind.

Neben dem einfachen Zusammenstellen des Datensatzes kann es jedoch auch vorkommen, dass Daten zuerst bearbeitet werden müssen, sodass ein-

zelne Details auffälliger sind. Das kann geschehen, indem beispielsweise ein Bild auf die relevanten Pixel reduziert wird. Dadurch lernt das Netz nicht nur besser, sondern auch schneller und mit einer geringeren Datenmenge als würden Daten nicht optimiert werden.

Sämtliche Anpassungen der Daten müssen einheitlich und auch bei der Anwendung des Netzes mit unbekannten Daten erfolgen.

Außerdem ist es für die bevorstehende Auswertung (1.3.3) wichtig, dass ebenfalls ein *Testdatensatz* erstellt wird. Sein Aufbau ist gleich wie der des Trainingsdatensatzes, es sind lediglich andere Daten enthalten.

1.3.2 Trainieren

Mit dem fertigen Trainingsdatensatz kann das neuronale Netz *trainiert* werden. Dafür existieren unterschiedliche Trainingsalgorithmen, die sich auch zwischen besonderen Aufbauten neuronaler Netze und deren spezifischen Aufgaben unterscheiden können.

Je nach Aufgabe oder spezifischen Struktur des Netzes können die Algorithmen unterschiedlich gut abschließen. Daher ist es lohnenswert, das Training nicht nur auf einen der Algorithmen zu beschränken, sondern mehrere mit Bezug auf die durchzuführende Aufgabe zu testen.

1.3.3 Auswertung und Anwendung

Nachdem das Training abgeschlossen wurde, ist es interessant zu wissen, mit welcher Genauigkeit oder Zuverlässigkeit eine Aufgabe ausgeführt wird. Dafür kommt der zuvor erstellte Testdatensatz (1.3.1) in Einsatz, mit dem die Anzahl der falschen bzw. unerwünschten Ausgaben ermittelt wird und der somit eine Einschätzung des Netzes möglich macht.

Für diese Auswertung ist es wichtig, dass der Testdatensatz nicht dieselben Daten enthält wie der Trainingsdatensatz, was simulieren soll, wie sich das Netz mit bislang unbekannten Daten verhält. Würde man das neuronale Netz mit den Trainingsdaten evaluiert werden, könnte es im schlechtesten Fall vorkommen, dass das Netz jedes kleinste Detail einer bekannten Eingabe anlernt, zu anderen Daten allerdings nahezu zufällige Ausgaben erzeugt. Eine solche Entwicklung wird auch *Overfitting* genannt. Dabei muss es sich jedoch nicht zwangsläufig um den hier beschriebenen Fall handeln, Overfitting kann auch eine Erscheinung sein, welche dann auftritt, wenn ein Training besonders viele Iterationen in Anspruch nimmt.

Wenn die Aufgabe zufriedenstellend abgeschlossen wird, können die Gewichte und die Konfiguration des Netzes in Dateien abgespeichert und in einem Anwendungsprogramm importiert. Das ermöglicht eine Rekonstruktion des neuronalen Netzes ohne großen Zeitaufwand und ohne erneut ein Training durchführen zu müssen.

2 Deep Convolutional Neural Networks [3]

Deep *Convolutional Neural Networks* (CNN) - im Deutschen *Faltungsnetze* - sind besondere Formen der neuronalen Netze, die besonders in der Bilderkennung und -Klassifikation sehr häufig verwendet werden. Grund dafür ist, dass sie viel schneller und besser trainiert werden können als herkömmliche neuronale Netze. Während diese eine Genauigkeit von über 90% erst spät erreichen, lernen CNN's innerhalb weniger Epochen ein Bild mit einer Wahrscheinlichkeit von über 95% korrekt zu klassifizieren.

Das ist auf deren Aufbau und Lernweise zurückzuführen, welche der des Menschen noch mehr ähnelt als die herkömmlicher neuronaler Netze.

2.1 Convolutional Layer

Verwendet man ein übliches neuronales Netz für die Bildklassifikation, ist die Relevanz zweier nebeneinanderliegender Pixel genauso hoch wie die von weit entfernten Pixeln. Das folgt daraus, dass die Ausgabe jedes Neurons an jedes Neuron der Folgeschicht propagiert wird.

In einem Convolutional Layer ist das anders: Mithilfe eines *Kernels* werden aus nebeneinanderliegenden Pixeln Eigenschaften erkannt. Erkennbare Eigenschaften beschränken sich in den höheren Schichten auf einfache Formen wie zum Beispiel Kanten, je tiefer sich eine Schicht jedoch im Netz befindet, kann diese immer komplexere Objekte anhand von Konturen unterscheiden.

Um das zu erreichen, wird die *Convolution*, oder eine Vereinfachung, die häufiger in Machine-Learning-Frameworks implementiert ist, die *Cross Correlation*, angewandt.

2.1.1 Eindimensionale Cross Correlation

Als *Cross Correlation* wird im Folgenden eine Operation mit einem Eingabevektor x und einem *Kernel* w betrachtet, aus dem ein Ausgabevektor

y berechnet werden soll. Die Cross Correlation ist eine Vereinfachung der *Convolution*, wobei der Unterschied im Wesentlichen darin besteht, dass der Kernel bei der Convolution zuerst gespiegelt werden muss.

Mit der Bedingung

$$|x| \geq |w| \qquad |x|,\ |w| > 0$$

kann die *valide* Cross Correlation im eindimensionalen Raum mathematisch durch die Formel

$$y = x \odot w \quad \rightarrow \quad y_i = \sum_{k=0}^{|w|-1} x_{i+k} * w_k \tag{2}$$

beschrieben werden. Im Folgenden entspricht das Symbol „$\odot$“ der Cross-Correlation-Operation. In Worten bedeutet das, dass der Kernel w an den Eingaben x angelegt wird. Addiert man das Produkt der einzelnen Werte x_i und w_i, ermittelt man y_i. Um y_{i+1} zu berechnen, muss der Kernel um eine Einheit verschoben werden. Ein Sinnbild ist in Abbildung 5 dargestellt.

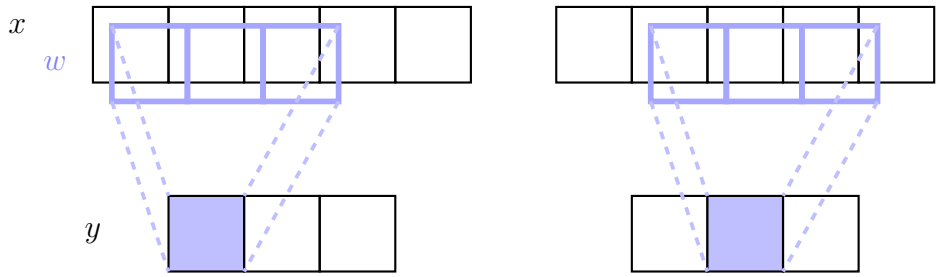


Abbildung 5: Cross Correlation im eindimensionalen Raum, ohne Padding

Es ist offensichtlich, dass die Länge des Ausgabevektors y unter diesen Bedingungen stets kleiner als die Länge der Eingabe sein wird. Ein solcher Informationsverlust von Pixeln kann jedoch zu Ungenauigkeiten führen. Daher wird ein *Padding* eingeführt, welches den Eingabevektor x künstlich mit Nullen erweitern soll. Zur Ermittlung des Ausgabevektors y entsteht das in Abbildung 6 dargestellte Schema. Man spricht nun nicht mehr von der validen Cross Correlation, sondern von einer Cross Correlation mit *Same Padding*, bei dem der Ausgabevektor y dieselbe Länge hat wie der Eingabevektor x. Darüber hinaus existiert noch das *Full Padding*, bei dem der Eingabevektor x mit noch mehr Nullen erweitert wird. Dadurch wird die Ausgabe nach

einem Convolutional Layer größer als die Eingabe, wodurch allerdings eine Vergrößerung von Daten stattfindet. Der damit verbundene Rechenaufwand ist der Grund, weshalb das Full Padding selten Anwendung findet.

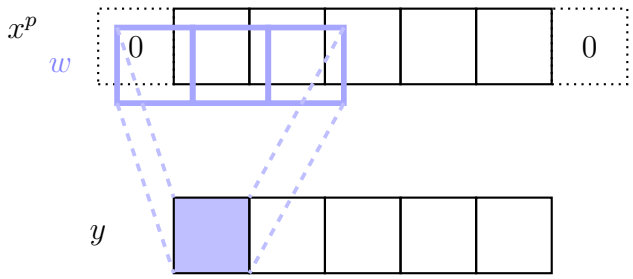


Abbildung 6: Cross Correlation im eindimensionalen Raum, mit Padding

2.1.2 Zweidimensionale Cross Correlation

Bilder sind allerdings nicht eindimensional, weshalb die zweidimensionale Cross Correlation benötigt wird.

Unter Definition der Variablen

- X^p: Eingabematrix ($\hat{=}$ Bild, einfarbig, mit Nullen entsprechend des Paddings erweitert)
- W: Kernel-Matrix
- Y: Ausgabematrix (Bild mit angewandtem Filter)

kann diese durch die Formel

$$Y = X \odot W \quad \rightarrow \quad Y_{i,j} = \sum_{k=0}^{|W|} \sum_{l=0}^{|W_l|} X^p_{i+k,j+l} * W_{k,j} \tag{3}$$

beschrieben werden, anhand des Schemas und Abbildung 7 ist sie jedoch einfacher zu verstehen.

Auch hier wird die Kernel-Matrix W an die Eingabematrix X^p angelegt und verschoben, um die Werte der Ausgabe Y durch Addieren der Produkte zu berechnen. Es ist jedoch zu beachten, dass W in diesem Fall in beide Richtungen verschoben wird, sodass eine zweidimensionale Ausgabe entsteht.

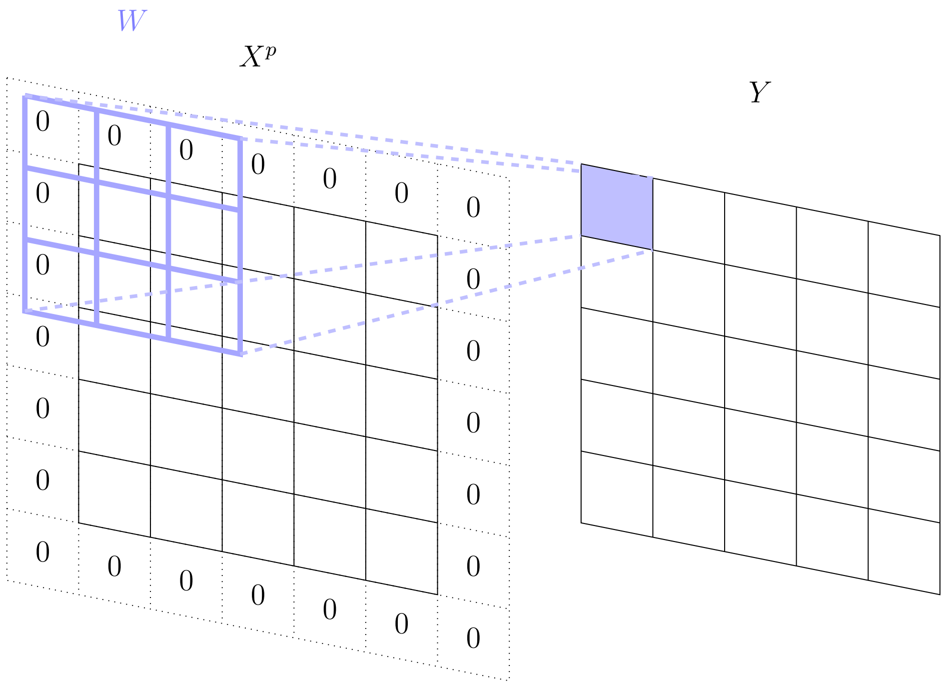


Abbildung 7: Cross Correlation im zweidimensionalen Raum

2.1.3 Praktische Anwendung

In der Praxis muss ein *Convolutional Layer* mehrere zweidimensionale Eingabematrizen annehmen können, da beispielsweise Bilder nicht (alle) einfarbig sind. Daher werden mehrere Eingabechannel definiert, für jede Eingabematrix einer. Um beispielsweise ein Farbbild einzugeben, würden 3 Channel benötigt, da ein solches Bild in die Farben rot, grün und blau unterteilt ist.

Auch die Anzahl der Ausgabematrizen soll für jede Schicht variabel, aber konstant sein. Jede Eingabematrix hat dabei einen Einfluss auf jede Ausgabematrix.

Um diesen Einfluss zu steuern wird eine Anpassung der Dimensionen des Kernels benötigt. Diese sind nun die Anzahl der Ausgabematrizen, die Anzahl der Eingabematrizen, die Größe und die Breite der Gewichte, weshalb der Kernel nun als vierdimensionales Array implementiert wird.

Eine Berechnung der Ausgabematrix Y_i des Ausgabechannels i lässt sich

nun formal durch

$$Y_i = \sum_{j=0}^{|X|-1} X_j \odot W_{i,j} \tag{4}$$

ausdrücken.

Im Anschluss an die Berechnung einer Ausgabematrix Y_i muss darauf noch die bereits aus Kapitel 1 bekannte differenzierbare Aktivierungsfunktion angewandt werden. Es entsteht mit

$$A_i = \varphi(Y_i)$$

eine sogenannte *Feature Map*. Diese enthalten in tieferen Schichten immer abstrakter werdende Eigenschaften der zu klassifizierenden Objekte, welche eine viel genauere Erkennung eines Objekts ermöglicht.

2.2 Subsampling Layer

Subsampling Layers kommen in neuronalen Netzen vor, um die Menge an Eigenschaften einer Eingabe mittels *Pooling-Operationen* zu verringern. Das resultiert in einem deutlich schnelleren Trainingsvorgang mit nur geringerem Leistungsverlust, da nach einem Subsampling Layer deutlich weniger Gewichte angepasst werden müssen.

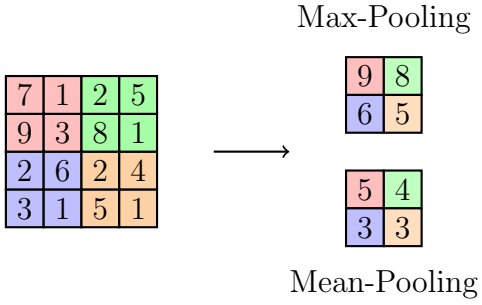


Abbildung 8: Max- und Mean-Pooling

Eine Eingabe X wird in mehrere kleine Pooling-Matrizen P aufgeteilt. Ab diesem Punkt können zwei unterschiedliche Arten angewandt werden: Bei *Max-Pooling* wird jeweils der größte Wert der Pooling-Matrix in die Ausgabe weitergeleitet, bei *Mean-Pooling* wird der Durchschnitt aller darin enthaltenen Werte ermittelt.

Subsampling Layer selbst haben keine Gewichte sondern beruhen auf simplen mathematischen Berechnungen wie dem Durchschnitt oder einem Größenvergleich.

2.3 Aufbau von CNN's

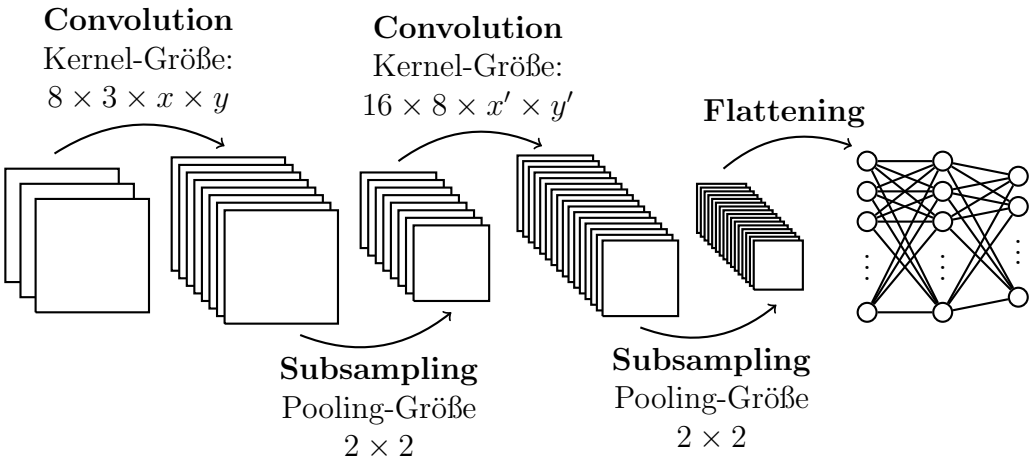


Abbildung 9: Exemplarischer Aufbau eines Convolutional Neural Networks

Convolutional Neural Networks sind sowohl aus diesen Convolutional und Subsampling Layers, als auch aus den bereits bekannten vollständig verbundenen Schichten aufgebaut.

Der im Folgenden beschriebene Aufbau ist in Abbildung 9 dargestellt.

Noch außerhalb des Netzes wird ein farbiges Bild in die drei Farbkomponenten rot, grün und blau aufgeteilt, sodass für jede eine einzelne Matrix entsteht mit Werten, die beschreiben, wie hoch der Anteil der jeweiligen Farbe an dem entsprechenden Pixel ist.

Die entstandenen Matrizen entsprechen der Eingabe des ersten Convolutional Layer. Diese Schicht führt Convolutions bzw. Cross Correlations mit einem Kernel der Größe 3×3 oder 5×5 durch, wobei die Anzahl der Ausgabechannel meist zwischen 8 und 32 liegt. Die optimale Wahl dieser Anzahl oder der Kernel-Größe ist abhängig von der auszuführenden Aufgabe und ihrer Komplexität sowie der Größe der Bilder. Am besten lassen sich die Werte in Tests ermitteln.

Anschließend erfolgt ein Subsampling der Matrizen mit der Pooling-Größe 2×2, handelt es sich jedoch um größere Bilder mit größeren Details können diese aber auch mit der Pooling-Größe 4×4 verkleinert werden. Im Normalfall wird die Max-Pooling-Strategie verwendet, auf einen Vergleich mit Mean-Pooling sollte jedoch nicht ohne Weiteres verzichtet werden.

Je nach Komplexität der zu erledigenden Aufgabe und der nach dem Pooling verbleibenden Größe der Feature Map folgen weitere Convolutions mit jeweils einem Subsampling Layer. Auch hier sind die einzelnen Werte der Schichten mit Feingefühl zu ermitteln und sollten zu Testzwecken verändert werden.

Nach der letzten Subsampling-Schicht erfolgt ein Abflachen oder *Flattening* aller Matrizen, was in der Regel in einer eigenen Schicht realisiert wird. Dieses Abflachen geschieht durch Aneinanderreihen der Zeilen aller Matrizen, wodurch letztendlich ein eindimensionaler Vektor entsteht.

Dieser Vektor kann durch ein herkömmliches, vollständig verbundenes neuronales Netz propagiert werden. Da die Eingabe sehr abstrakte Eigenschaften beschreibt, die unmittelbar mit der zuzuordnenden Klasse in Verbindung stehen, wird hier in den meisten Fällen nicht mehr als eine versteckte Schicht benötigt.

2.4 Ziffernerkennung mit einem Convolutional Neural Network

2.4.1 Der MNIST-Datensatz

Beim MNIST-Datensatz handelt es sich um einen einfachen Bilderkennungs-Datensatz, der wegen seiner Simplizität und Bekanntheit gerne als Einstiegs- oder Testbeispiel verwendet wird.

Er umfasst 70.000 Graustufen-Bilder der Größe 28×28 Pixel, wobei jedes Bild eine handgeschriebene Ziffer von 0 bis 9 enthält. Das Ziel des Netzes soll es sein, die Bilder eindeutig einer der Ziffern zuzuordnen.

2.4.2 Das TensorFlow-Framework

TensorFlow ist ein Open-Source Machine-Learning-Framework von Google, das hauptsächlich auf die Programmiersprachen Python und C++ ausgelegt ist, jedoch auch mit vielen weiteren kompatibel ist. Mit der Version 2.0

werden mittlerweile nahezu alle Architekturen neuronaler Netze unterstützt, wodurch es im Bereich des Data Science sehr beliebt wurde.

Mit TensorFlow hat man die Möglichkeit, sein eigenes neuronales Netz mit selbst gewählten Parametern Schicht für Schicht zusammenzustellen. Das ermöglicht jedem, selbst einen idealen Aufbau zu finden, der die Aufgabe am Besten erfüllt. Das Training, und somit der schwierigere Teil, wird allerdings vom Framework abgenommen.

2.4.3 Ziffernerkennung

Im Folgenden soll TensorFlow dazu verwendet werden, die Daten des MNIST-Datensatzes mit einem CNN zu klassifizieren. Anschließend erfolgt ein Vergleich der erreichten Ergebnisse mit denen eines vollständig verbundenen neuronalen Netzes.

Sämtlicher Code kann unter

`https://colab.research.google.com/drive/1n-t1O-3lq-VovD4DCZmhpc9q8Ht4n4At`

abgerufen werden. Alternativ befindet sich am Ende des Kapitels ein QR-Code zum Scannen.

Laden des MNIST-Datensatzes in TensorFlow

Mit dem folgenden Code wird das TensorFlow-Framework eingebunden und der MNIST-Datensatz heruntergeladen. Im Anschluss werden die Farbanteile von einem ursprünglichen Bereich von 0 bis 255 auf 0 bis 1 skaliert und eine Daten in Trainings- und Testdatensatz eingeteilt, jeweils mit einer Batch-Größe von 64 Daten. Das bedeutet, dass mit einer Trainings-Iteration 64 Daten trainiert werden.

```
import tensorflow as tf
import tensorflow_datasets as tfds

## Build Datasets
mnist_builder = tfds.builder('mnist')
mnist_builder.download_and_prepare()
datasets = mnist_builder.as_dataset(shuffle_files=False)
mnist_train_data = datasets['train']
mnist_test_data = datasets['test']

mnist_train = mnist_train_data.map(
    lambda item:
    (tf.cast(item['image'], tf.float32)/255.0,
     tf.cast(item['label'], tf.int32))
```

```
)
mnist_test = mnist_test_data.map(
    lambda item:
    (tf.cast(item['image'], tf.float32)/255.0,
     tf.cast(item['label'], tf.int32))
)
tf.random.set_seed(1)
mnist_train = mnist_train.shuffle(
    buffer_size=64,
    reshuffle_each_iteration=False
)

mnist_validate = mnist_test.take(10000).batch(64)
mnist_train = mnist_train.skip(10000).batch(64)
```

Aufbau des Netzes

Das neuronale Netz soll zunächst gleich aufgebaut sein wie in Abbildung 9 dargestellt. Dafür werden jeweils zwei Convolutional- und Subsampling-Layer benögigt, gefolgt von einem Flattening Layer und zwei vollständig verbundenen Schichten.

```
## Build Model
model = tf.keras.Sequential()

## Convolutional and Pooling Layers
model.add(tf.keras.layers.Conv2D(
    filters=8,
    kernel_size=(3, 3),
    strides=(1, 1),
    padding='same',
    data_format='channels_last',
    name='conv1',
    activation='relu'
))
model.add(tf.keras.layers.AveragePooling2D(
    pool_size=(2, 2),
    name='pool1'
))
model.add(tf.keras.layers.Conv2D(
    filters=16,
    kernel_size=(3, 3),
    strides=(1, 1),
    padding='same',
    data_format='channels_last',
    name='conv2',
    activation='relu'
))
model.add(tf.keras.layers.AveragePooling2D(
    pool_size=(2, 2),
    name='pool2'
))
```

```
## Flattening Layer
model.add(tf.keras.layers.Flatten())

## Fully Connected Layers
model.add(tf.keras.layers.Dense(
    units=1024,
    name='fc1',
    activation='relu'
))
model.add(tf.keras.layers.Dense(
    units=10,
    name='fc2',
    activation='softmax'
))

## Compile the Model
tf.random.set_seed(1)
model.build(input_shape=(None, 28, 28, 1))
model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)
```

Training

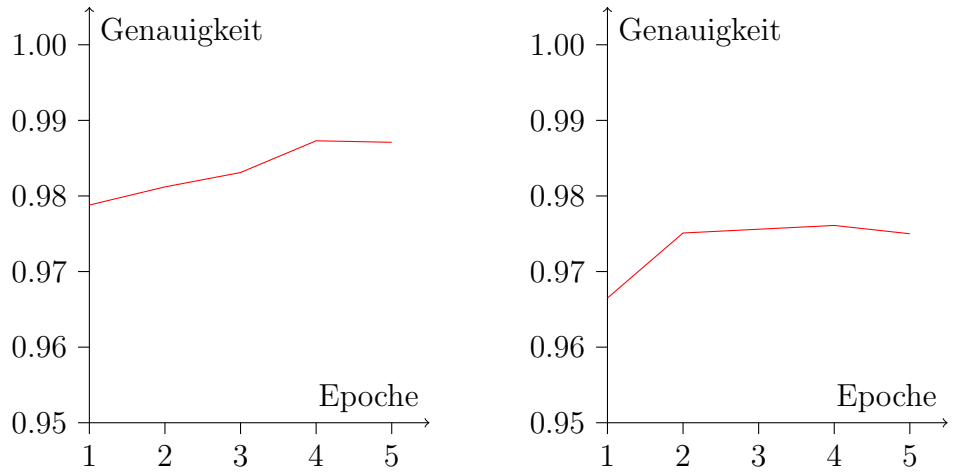
Das erzeugte neuronale Netz wird anschließend mit dem Methodenaufruf

```
model.fit(
    mnist_train,
    epochs=5,
    validation_data=mnist_validate
)
```

trainiert. Die Anzahl der Trainingsiterationen, hier *Epoche* genannt, ist dabei auf 5 begrenzt.

Ergebnisse

In Abbildung 10 sind die Trainingsverläufe dieses CNN's und eines herkömmlichen neuronalen Netzes über den Verlauf der fünf Epochen dargestellt. Es fällt auf, dass das herkömmliche neuronale Netz immer schlechter abschneidet als das CNN, auch nicht, wenn noch mehr Epochen erlaubt werden. Es sei jedoch angemerkt, dass eine hier erreichte Genauigkeit von etwa 98% für ein einfaches, vollständig verbundenes neuronales Netz äußerst zufriedenstellend ist. Üblich sind Werte um etwa 90%.



(a) CNN (b) FCNN

Abbildung 10: Trainingsverlauf eines CNN's (10a) im Vergleich zu einem herkömmlichen neuronalen Netz (10b)

Code

Der folgende Link verweist auf ein Notebook in Google Colab, welches u. A. das Ausführen von Python-Code in der Cloud ermöglicht.

`https://colab.research.google.com/drive/1n-t1O-3lq-VovD4DCZmhpc9q8Ht4n4At`

3 Resultate

Mit einem künstlichen neuronalen Netz wird einem Computerprogramm ermöglicht, wie der Mensch zu denken oder Entscheidungen zu treffen, wobei ein Netz jedoch meistens nur auf eine bestimmte Aufgabe ausgelegt ist. Für einzelne Aufgaben existieren ebenfalls unterschiedliche Architekturen neuronaler Netze, für die Bilderkennung verwendet man beispielsweise Faltungsnetze, welche die relevanten Informationen eines Bildes extrahieren. Objekte können dadurch ähnlich gut klassifiziert werden wie von einem Menschen (vgl. Kapitel 2.4.2).

Aufgrund dieser Genauigkeit haben neuronale Netze auch dort großes Potenzial, wo Fehler auch schwere Folgen haben können:

Nahezu die einzige Entscheidungsgrundlage eines selbstfahrenden Autos sind die Aufnahmen von Kameras. Um rechtzeitig zu lenken, bremsen oder zu beschleunigen müssen die Bilder innerhalb von Bruchteilen einer Sekunde ausgewertet werden. Oft ist eine Entscheidung des Fahrzeugs besonders in Gefahrensituationen sicherer, da alle Kameras gleichzeitig ausgewertet werden und nicht nur die Kamera in eine Richtung. Außerdem ist eine Berechnung weiter Informationen möglich, zum Beispiel ob ein Bremsweg ausreicht oder ob ein Ausweichmaneuver nötig ist.

Auch im medizinischen Bereich unterstützen neuronale Netze Ärzte bei beispielsweise der Auswertung bildgebender Verfahren. Mögliche Entscheidungskriterien, zum Beispiel Unregelmäßigkeiten, werden dabei hervorgehoben um die Diagnose zu erleichtern. Auch dafür eignen sich Faltungsnetze. [5]

Literatur

[1] Callan, Robert: *Neuronale Netze im Klartext.* Pearson Studium, 1. Auflage, 2003.

[2] Raschka, Sebastian und Vahid Mirjalili: *Python Machine Learning.* Packt, 3. Auflage, 2015. Kapitel 1: Giving Computers the Ability to Learn from Data.

[3] Raschka, Sebastian und Vahid Mirjalili: *Python Machine Learning.* Packt, 3. Auflage, 2015. Kapitel 15: Classifying Images with Deep Convolutional Networks.

[4] Spitzer, Manfred: *The Mind Within the Net: Models of Learning, Thinking and Acting*, Seite 21. MIT Press, 2. Auflage, 1999. `https://books.google.de/books?id=lniTl1DbRX4C&pg=0`. Stand: 23.04.2022.

[5] Wikipedia: *Künstliche Intelligenz in der Medizin.* `https://de.wikipedia.org/w/index.php?title=K%C3%BCnstliche_Intelligenz_in_der_Medizin&oldid=223677378`. Stand: 16.06.2022.

Index