

Theodor-Heuss-Gymnasium Göttingen

Lehrer: Herr Dr. Dillmann

Informatik

Bilderkennung mit künstlicher Intelligenz

Theorie des Convolutional Neural Network und
Implementierung in Python

Seminarfacharbeit

von David Söding

23. März 2022

Inhaltsverzeichnis

1 Einleitung	2
2 Künstliche neuronale Netzwerke	4
2.1 Grundlagen des künstlichen neuronalen Netzes	4
2.2 Beispiel für ein einfaches KNN	6
2.3 Optimierung und Training	8
2.3.1 Verlustfunktionen	9
2.3.2 Gradientenverfahren	12
2.4 Funktionsweise und Besonderheiten des CNN	15
2.4.1 Convolutional Layer	15
2.4.2 Pooling Layer	16
2.4.3 Fully connected Layer	17
3 Entwicklung eines eigenen CNNs in Python	18
3.1 Implementierung	18
3.1.1 Installation und Setup	19
3.1.2 Erstellen des Modells	20
3.1.3 Weitere Verbesserungen	25
3.2 Evaluierung des CNN	29
4 Abschließende Bemerkungen	33

1 Einleitung

Schon lange träumen Wissenschaftler von Maschinen, die eigenständig Probleme lösen können und zum eigenen Denken und Lernen in der Lage sind; Programme, die man nicht programmieren muss und Algorithmen, die sich von alleine auf ihre Aufgaben einstellen. Dies alles ist dank der wissenschaftlichen Entwicklungen des letzten Jahrzehnts keine Fantasie mehr. Heute ist künstliche Intelligenz nicht nur ein großer Teil von vielen wissenschaftlichen Gebieten wie Biologie und Medizin, sondern auch Teil des Alltags. Ob Apples Face-ID oder Teslas Autopilot, vieles wird heute durch künstliche Intelligenz und maschinelles Lernen ermöglicht. Maschinelles Lernen ist der Teil der künstlichen Intelligenz, welcher das selbstständige Lernen und Anpassen von Algorithmen umfasst. Dass Algorithmen nicht mehr speziell programmiert werden müssen, um ein Problem zu lösen, ist eine technische Meisterleistung. Stattdessen übt der Algorithmus seine Aufgabe an Trainingsdaten und lernt dadurch, diese Aufgabe auch bei neuen Daten auszuführen.

Von den Neuronen und Synapsen des menschlichen Gehirns inspiriert, wurden künstliche neuronale Netze entwickelt. Diese sind Rechensysteme, die aus einem Netz verbundener künstlicher Neuronen bestehen. Jede Verbindung zweier dieser künstlichen Neuronen kann, wie Synapsen in einem echten Gehirn, ein Signal weiterleiten. Indem die Stärke dieser Verbindungen systematisch verändert wird, kann ein komplexes Netz an Neuronen erschaffen werden, welches, basierend auf verschiedenen Eingabedaten, unterschiedliche Neuronen aktiviert und so ein gewünschtes Ergebnis ausgibt.

Der Aufgabenbereich der künstlichen neuronalen Netze, der im Fokus dieser Facharbeit liegt, ist die Bilderkennung. Ziel ist es, Objekte in Bildern erkennen und kategorisieren zu können. Dies kann für eine Vielzahl von Anwendungsbereichen sinnvoll sein. Unter anderem wurde mit Hilfe von Röntgenbildern der weiblichen Brust vorhergesagt, welche Patientinnen ein erhöhtes Risiko haben, an Brustkrebs zu erkranken. Hierfür wurden 89 000 Scans von 40 000 Frauen, welche über einen Zeitraum von vier Jahren untersucht wurden, gesammelt. An einem Teil dieses Datensatzes wurde der Algorithmus dann trainiert. An dem anderen Teil wurde er getestet, wobei der Algorithmus 31% der Frauen, die später an Brustkrebs erkrankten, in die Hoch-Risiko-Gruppe einteilte. Bisherige Methoden, welche auf Faktoren wie Alter und Familiengeschichte von Krebs basieren, haben nur 18% dieser Frauen der Gruppe zugeteilt [13].

Im ersten Teil dieser Facharbeit werden die Grundlagen von neuronalen Netzen ausführlich erläutert und beschrieben. Es wird auf die einzelnen Bausteine eines solchen Netzes, die Neuronen und ihre Verbindungen eingegangen, sowie die Optimierung und das Training an einfachen Beispielen erklärt. Zusätzlich wird die Funktionsweise von Convolutional Neural Networks (CNNs) beschrieben, eine spezielle Art der

neuronalen Netze, die sich insbesondere für Bilderkennung gut eignet.

Im zweiten Teil der Facharbeit wird, basierend auf den zuvor dargelegten Informationen, Schritt für Schritt erklärt, wie sich ein CNN modellieren, programmieren und trainieren lässt, um Objekte wie Tiere und Autos in Bildern zu erkennen. Dafür wird die Programmiersprache Python benutzt, da sie leicht zu erlernen ist und eine Vielzahl an hilfreichen Bibliotheken zur Verfügung steht. Anschließend wird das fertige Modell analysiert und getestet, um daraus Schlüsse über das Training und die Funktionsweise der neuronalen Netzwerke zu ziehen.

2 Künstliche neuronale Netzwerke

Um den Aufbau eines solchen Netzwerkes genauer zu verstehen, werden im folgenden Abschnitt die grundlegenden Begriffe erklärt und die Funktionsweise eines künstlichen neuronalen Netzes (KNN) an einem einfachen Beispiel erläutert. Anschließend werden Begriffe eingeführt, die für das Verständnis der Optimierung und des Trainings eines neuronalen Netzes notwendig sind. Zuletzt wird ein Einblick in die konkreten Bausteine eines Convolutional Neural Network gegeben, mit deren Hilfe in Kapitel 3 die Implementierung nachvollzogen werden kann.

In dieser Facharbeit werden ausschließlich Feed-Forward-Netzwerke behandelt, in denen Informationen stets in eine Richtung fließen und somit keine Rückkopplung oder Ähnliches stattfindet.

2.1 Grundlagen des künstlichen neuronalen Netzes

- **Neuronen** im KNN sind Knoten mit einem bestimmten Wert, die sich durch Verbindungen beeinflussen können. Durch diese Verbindungen beeinflussen sie die darauf folgenden Neuronen der nächsten Ebene. Die Eingabe sowie die Ausgabe eines KNN besteht auch aus Neuronen.
- **Gewichte** bestimmen die Relevanz der einzelnen Verbindungen. Sie geben in Form eines Faktors an, welchen Einfluss die Eingabe auf die Ausgabe hat.
- Jedes Neuron besitzt einen **Bias**. Der Bias ist ein Wert, der das Neuron, unabhängig von den Verbindungen, beeinflusst.
- Ein **Layer** oder eine Ebene besteht aus mehreren Neuronen. Jedes dieser Neuronen hat die gleiche Funktion und ist nicht mit Neuronen des gleichen Layers verbunden. Jeder Layer beeinflusst ausschließlich den darauf folgenden Layer [5, S.163]. Der erste Layer eines KNN ist immer der Eingabe- und der letzte der Ausgabe-Layer. Alle Layer dazwischen werden Hidden-Layer genannt, da deren Werte normalerweise nicht sichtbar sind.
- **Aktivierungsfunktionen** sind Funktionen, die auf den Wert eines Neurons angewandt werden, um dem KNN Nichtlinearität verleihen. Ein neuronales Netz ohne Aktivierungsfunktion kann nur lineare Probleme lösen, ähnlich wie lineare Regression. In modernen neuronalen Netzen wird als Aktivierungsfunktion häufig die ReLU (rectified linear unit) verwendet. Diese Aktivierungsfunktion hat sich in den letzten Jahren durchgesetzt, da sie effizient zu berechnen ist und aus zwei linearen Teilen besteht, weshalb die einfache Optimierung mit Gradienten-basierten Methoden linearer Modelle erhalten bleibt [5, S.187]. Auch nennenswert ist die Sigmoid Funktion sowie der Tangens hyperbolicus.

Diese schränken die Funktionswerte auf ein Intervall von 0 bis 1 beziehungsweise -1 bis 1 ein, werden heutzutage aber wenig benutzt, da sie das aufwändige Errechnen von Werten von Exponentialfunktionen erfordern [5, S.189].

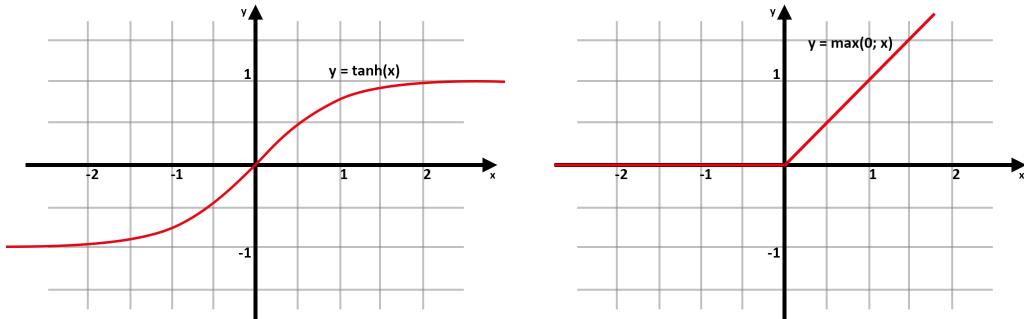


Abb. 1: Links: Tangens hyperbolicus; rechts: ReLU – rectified linear unit. ©David Söding, 2022

Der Ausgabewert h_i von Neuron i ergibt sich durch

$$h_i = f\left(\left(\sum_{j=1}^N W_{ij}x_j\right) + b_i\right) \quad ,$$

wobei N die Anzahl an Eingabe-Neuronen, x_j der Wert jedes einzelnen Eingabe-Neurons und W_{ij} das Gewicht jeder Verbindung eines Eingabe-Neurons zu i ist. Der Bias b_i von i wird addiert, bevor die Aktivierungsfunktion angewandt wird. Er verschiebt also, unabhängig von den Verbindungen des Neurons, die Eingabe zur Aktivierungsfunktion f [14, S.81].

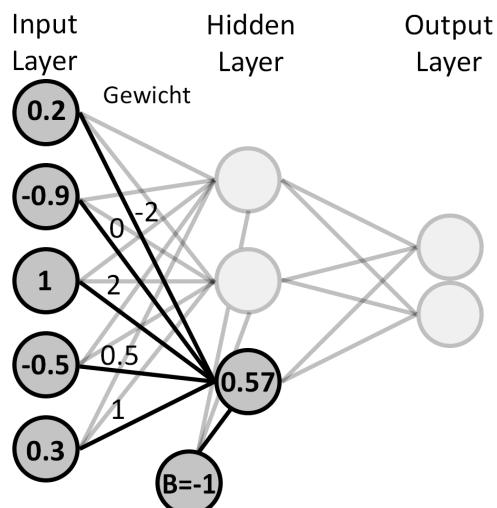


Abb. 2: Beispiel der Beeinflussung von Eingabe-Neuronen, Gewichtung und Bias auf ein Neuron. ©David Söding, 2022

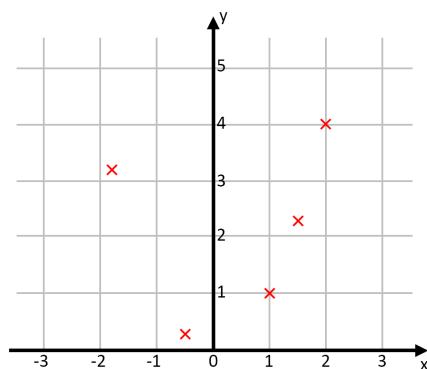
In dem Beispiel aus Abbildung 2 besteht der Eingabe-Layer aus fünf Neuronen mit Werten zwischen -1 und 1 und dem Bias. Jeder Wert x_i , $i = 1, \dots, 5$, wird mit dem

jeweiligen Gewicht $W_{i,j}$, $i = 1, \dots, 5$ multipliziert. Anschließend werden alle Werte und der Bias addiert und in die Aktivierungsfunktion (hier Tangens hyperbolicus) eingesetzt. Der Tangens hyperbolicus von 0.65 ergibt gerundet 0.57, weshalb das Neuron dies als neuen Wert annimmt.

2.2 Beispiel für ein einfaches KNN

Ein künstliches neuronales Netz ist nichts weiter als eine komplizierte mathematische Funktion, die jeder Eingabe eine bestimmte Ausgabe zuweist. Wie dies funktioniert, wird im folgenden Abschnitt genau erläutert. Dafür wird hier erst einmal das Trainieren eines solchen Netzes übersprungen.

Wie in Abbildung 3 und Tabelle 1 zu sehen ist, wurde in einem beliebigen Experiment das Verhältnis zweier Variablen x und y untersucht. Das Ziel des KNN ist es nun, basierend auf den gegebenen Messwerten, die y -Werte von noch nie zuvor gesehenen x -Werten vorherzusagen.



x	y
-1.8	3.24
-0.5	0.25
1	1
1.5	2.25
2	4

Abb. 3: Messwerte im Koordinatensystem.
©David Söding, 2022

Tabelle 1: Messwerte

Dafür wird die ReLU-Einheit verwendet. Da nur ein Eingabe-Wert x und ein Ausgabe-Wert y existiert, lässt sich das KNN in einem zweidimensionalen Koordinatensystem darstellen. KNNs können aber beliebige Eingabe- und Ausgabe-Dimension haben.

Die Funktion f in Abbildung 4 ist ein einfaches ReLU. Die Funktion g stellt ein Neuron dar, welches mit der Eingabe über ein Gewicht von 2 verbunden ist, und zusätzlich einen Bias von -2 hat. Das Gewicht verändert hier die Steigung der Funktion, während der Bias den Graphen nach rechts verschiebt.

Werden $f(x)$ und $g(x)$ addiert, so ergibt sich die Funktion h . Diese Funktion hat bereits zwei Knickstellen. Hier offenbart sich das Problem eines linearen KNN. Werden zwei beliebige lineare Funktionen addiert, so ergibt sich immer eine neue lineare Funktion ohne Knickstellen, egal wie die Steigung oder Verschiebung beider Funktionen sich unterscheidet. Dadurch lässt sich keine nichtlineare Funktion bilden, weshalb durch eine Aktivierungsfunktion Nichtlinearität hinzugefügt wird.

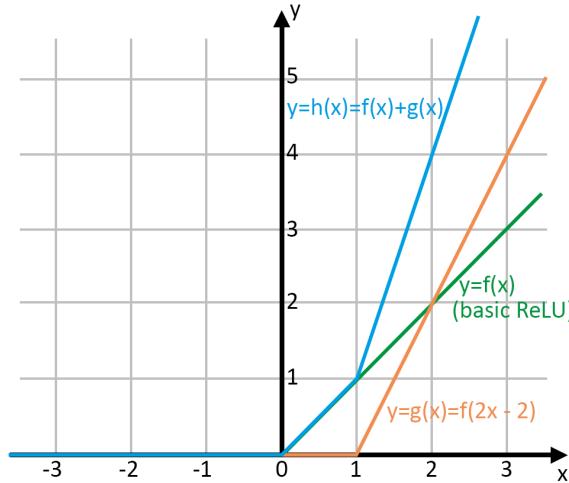


Abb. 4: Addition verschiedener ReLU. ©David Söding, 2022

Auf diese in Abbildung 4 dargestellte Weise lässt sich aus einfachen Bausteinen eine komplexe Funktion erstellen. Um das Beispiel einfach zu halten, wurde hier kein wirkliches KNN trainiert. Das folgende Modell ist nur eine grobe und vereinfachte Schätzung, wie ein trainiertes Modell aussehen könnte. Außerdem ist der Trainingsdatensatz viel zu klein, um ein präzises Ergebnis zu erhalten, weshalb es nur zur Veranschaulichung dient.

Wie sich in Abbildung 5 zeigt, besteht das KNN aus einem Hidden-Layer mit 6 Neuronen. Die Summe dieser sechs ReLU-Funktionen ergibt den Graphen aus Abbildung 6. Das KNN lässt sich nun als mathematische Funktion ausdrücken. Angenommen $f(x) = \text{ReLU}(x)$ (siehe Abbildung 4), so ist die Funktion des KNN

$$k(x) = f(x) + f(2x - 2) + f(4x - 8) + f(-x) + f(-2x - 2) + f(-4x - 8),$$

wobei die Faktoren vor x das Gewicht und die subtrahierten Werte der Bias der einzelnen Neuronen sind. Dies lässt sich auch mit Summenzeichen allgemeiner formulieren durch

$$k(x) = \sum_{j=1, j'=1}^6 W_{j'} f(W_j x + b_j) ,$$

wobei W_j das Gewicht des jeweiligen Neurons, b_j der Bias und x die Eingabe ist. $W_{j'}$ ist das Gewicht der Verbindung des Neurons im Hidden-Layer zur Ausgabe.

Wie sich in Abbildung 6 erkennen lässt, ist das nun trainierte KNN gut auf die Trainingsdaten abgestimmt. Jetzt gilt es zu überprüfen, ob das KNN auch das zuvor gesetzte Ziel erreicht hat. Kann es einem bislang unbekannten Wert x einen Wert y zuordnen?

Wird dem KNN die Eingabe 1.8 gegeben, so ist die Ausgabe $k(1.8) = 3.4$ (siehe

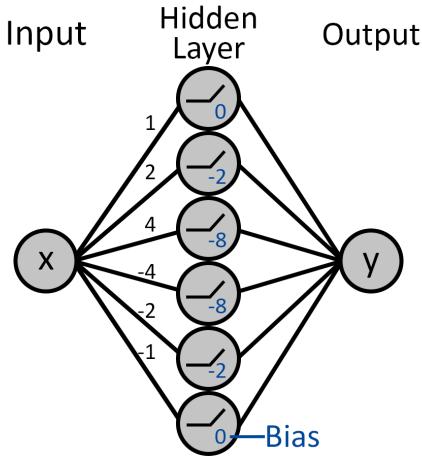


Abb. 5: Modell des KNN. ©David Söding, 2022

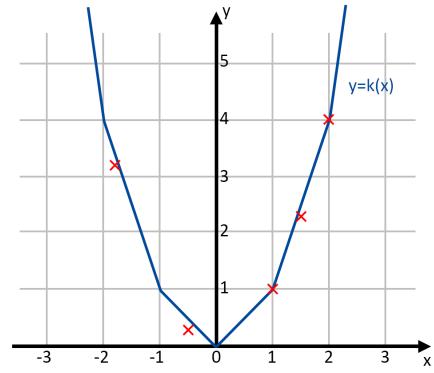


Abb. 6: Funktionsgraph des KNN. ©David Söding, 2022

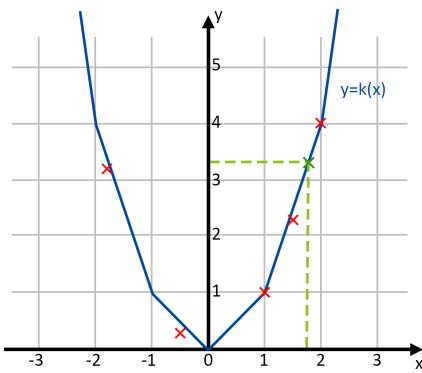


Abb. 7: Schätzung des KNN. ©David Söding, 2022

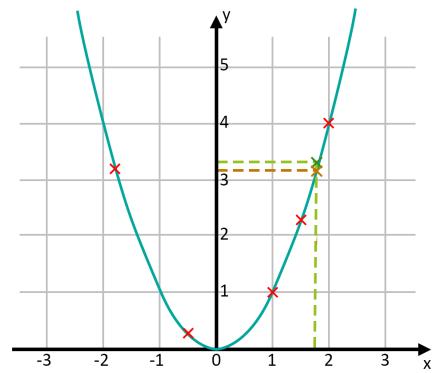


Abb. 8: Tatsächliche Genauigkeit des KNN. ©David Söding, 2022

Abbildung 7). Da die Testwerte zu Demonstrationszwecken auf der Normalparabel liegen, lässt sich das Ergebnis leicht überprüfen. Wie sich in Abbildung 8 erkennen lässt, hat das Ergebnis eine akzeptable Genauigkeit. Der korrekte y -Wert für $x = 1.8$ ist $1.8^2 = 3.24$, was eine Abweichung von 0.16 von der Berechnung des KNN ist.

Dieses einfache KNN, welches nur einen Hidden-Layer besitzt, hat die Gewichte und den Bias der Neuronen mit Hilfe der Trainingsdaten angepasst. Es war am Ende in der Lage, einem x -Wert, den es zuvor im Trainingsdatensatz nie gesehen hat, einen y -Wert mit akzeptabler Genauigkeit zuzuordnen.

2.3 Optimierung und Training

Da im vorherigen Abschnitt das Trainieren eines künstlichen neuronalen Netzes übersprungen wurde, wird nun genauer auf diesen Vorgang eingegangen.

2.3.1 Verlustfunktionen

Um ein neuronales Netz optimieren zu können, ist eine Verlustfunktion notwendig. Diese bewertet, wie gut das KNN beim Lösen einer Aufgabe auf einem Trainingsdatensatz abschneidet. Wird der Wert der Verlustfunktion kleiner, bedeutet dies, dass die Genauigkeit des neuronalen Netzes sich auf dem Datensatz verbessert [5, S.79]. Um die Verlustfunktion zu minimieren, müssen die Parameter des KNN verändert werden. Parameter beim Lernen des KNN sind die Gewichte und Biases. Werte wie Anzahl der Layer oder Anzahl der Neuronen werden nicht automatisch angepasst und heißen deshalb Hyperparameter. Hierfür muss manuell ein passendes Modell gefunden werden. Werden die Parameter des KNN geändert, verändert sich auch der Wert der Verlustfunktion. Die Variablen der Verlustfunktion sind demnach alle Parameter des KNN. Lässt sich das Minimum der Verlustfunktion finden, so weiß man, welche Parameter das KNN benötigt, um optimale Ergebnisse zu erzielen. Wie eine Verlustfunktion aufstellt und minimiert wird, lässt sich am Beispiel der linearen Regression erläutern.

Die gegebenen Messwerte in Abbildung 9 und Tabelle 2 stehen für ein beliebiges Beispiel. In diesem Fall sind es die Preise von Grundstücken relativ zu ihrer Größe. Soll nun herausgefunden werden, für welchen Preis ein Grundstück mit einer Fläche von 2500m^2 verkauft werden kann, gibt es für diesen x-Wert keine Daten. Also wird ein neuronales Netz trainiert, welches basierend auf den Trainingsdaten in Tabelle 2 mit Hilfe linearer Regression vorhersagt, für wie viel Geld ein Grundstück verkauft werden kann.

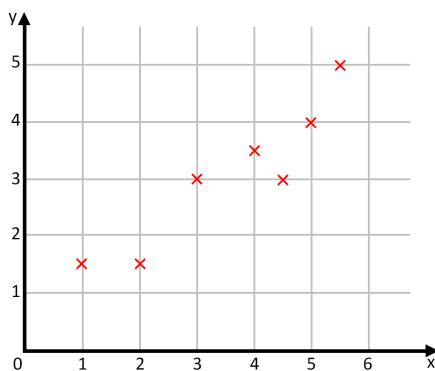


Abb. 9: Grundstücksfläche auf der x-Achse in 1000m^2 und Grundstückpreise auf der y-Achse in $100\,000\text{€}$. ©David Söding, 2022

Grundstücksfläche in 1000m^2	Grundstückpreise in $100\,000\text{€}$
1	1.5
2	1.5
3	3
4	3.5
4.5	3
5	4
5.5	5

Tabelle 2: Messwerte

Ein Trainingssample, also ein Wertepaar von x- und y-Werten, besteht aus einer Variable und einem Zielwert. Die Variablen sind in diesem Fall die Grundstücksflächen, also die Eingaben, während die Grundstückspreise die Zielwerte, also die zu errechnenden Funktionswerte, sind.

Jede lineare Funktion lässt sich beschreiben durch $f(x) = mx + b$. Da m als Faktor

vor der Eingabe x steht, lässt sich m als Gewicht w verstehen. Weil b unabhängig von x addiert wird, kann b als Bias aufgefasst werden. Dadurch lässt sich $f(x)$ definieren als

$$f(x) = wx + b \quad .$$

Für die lineare Regression ist nur ein einziges Neuron zusätzlich zur Eingabe und Ausgabe notwendig. Um die Genauigkeit der Regression zu maximieren, muss eine Funktion definiert werden, die für die Eingabe w und b eine Genauigkeit angibt. Es wird eine Verlustfunktion benötigt. Eine Möglichkeit, die Genauigkeit des KNN zu bestimmen, ist die mittlere quadratische Abweichung zu berechnen. Diese gibt die durchschnittliche quadrierte Differenz von Vorhersage und Zielwert im Trainingsdatensatz an. Dies ist für diesen Zweck sinnvoll, da die Quadrierung alle Zahlen positiv macht und größere Abweichungen stärker gewichtet [5, S.105]. Sie ist außerdem leicht zu berechnen. Wenn x_i die x -Werte und y_i die jeweiligen Zielwerte des Trainingsdatensatzes sind, $f(x_i)$ die Vorhersage des KNN für x_i ist und m die Anzahl an Trainingsdaten-Paaren ist, so ist die mittlere quadratische Abweichung

$$\text{MQA} = \frac{1}{m} \sum_{i=1}^m (f(x_i) - y_i)^2,$$

vgl. [5, S.105]. Wird nun $f(x)$ mit der Funktion des neuronalen Netzes ersetzt, so erhält man

$$\text{MQA}(w, b) = \frac{1}{m} \sum_{i=1}^m ((wx_i + b) - y_i)^2.$$

Nun lässt sich die Abweichung von der Vorhersage des Netzes zu den Trainingsdaten relativ zu w und b berechnen. Da es zwei Eingabe-Variablen und eine Ausgabe gibt, lässt sich diese Funktion in einem dreidimensionalen Koordinatensystem darstellen (siehe Abbildung 10). Wird nun das Minimum dieser Funktion gefunden, so ist die Abweichung auch minimiert und die lineare Regression beendet. Um das Minimum der Funktion zu finden, wird sie partiell nach w und b abgeleitet. Bei einer partiellen Ableitung mit mehreren Argumenten werden die jeweils anderen Argumente als Konstanten betrachtet. Durch einige einfache Umformungen und Anwendung der

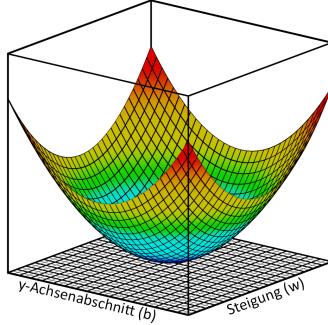


Abb. 10: Beispiel eines dreidimensionalen Graphen einer quadratischen Funktion mit zweidimensionalem Definitionsbereich. ©David Söding, 2022

Kettenregel ergeben sich:

$$\begin{aligned}
 \frac{\partial}{\partial b} \text{MQA}(w, b) &= \frac{\partial}{\partial b} \frac{1}{m} \sum_{i=1}^m ((wx_i + b) - y_i)^2 \\
 &= \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial b} ((wx_i + b) - y_i)^2 \\
 &= \frac{2}{m} \sum_{i=1}^m ((wx_i + b) - y_i) \\
 \frac{\partial}{\partial w} \text{MQA}(w, b) &= \frac{\partial}{\partial w} \frac{1}{m} \sum_{i=1}^m ((wx_i + b) - y_i)^2 \\
 &= \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial w} ((wx_i + b) - y_i)^2 \\
 &= \frac{2}{m} \sum_{i=1}^m ((wx_i + b) - y_i) x_i
 \end{aligned}$$

Anschließend werden beide Ableitungen gleich null gesetzt und nach der jeweiligen Variable aufgelöst, um das Minimum zu finden.

$$\begin{aligned}
 0 = \frac{\partial}{\partial b} \text{MQA}(w, b) &= \frac{2}{m} \sum_{i=1}^m ((wx_i + b) - y_i) \\
 b &= \frac{1}{14}(43 - 50w) \\
 0 = \frac{\partial}{\partial w} \text{MQA}(w, b) &= \frac{2}{m} \sum_{i=1}^m ((wx_i + b) - y_i) x_i \\
 w &= \frac{1}{211}(177 - 50b)
 \end{aligned}$$

Nun lässt sich eines der Ergebnisse in das andere einsetzen, wodurch sich die Werte

von w und b ergeben.

$$w = \frac{1}{211}(177 - 50(\frac{1}{14}(43 - 50w)))$$

$$w \approx 0.72247$$

$$b \approx \frac{1}{14}(43 - 50 \cdot 0.72247)$$

$$b \approx 0.49119$$

Dadurch, dass das Minimum der Verlustfunktion erreicht wurde, hat das neuronale Netz nun die optimalen Parameter. Die Funktion $f(x)$ des KNN ist nun also

$$f(x) = 0.72247x + 0.49119.$$

Jetzt kann das KNN auch vorhersagen, wie viel unser 2500m^2 Grundstück ungefähr wert ist. $f(2.5) \approx 2.30$, also könnte einen Preis von 230 000 € erwartet werden.

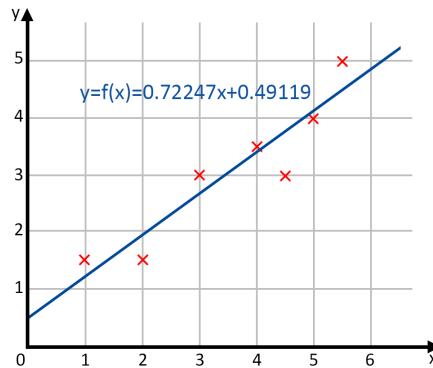


Abb. 11: Funktionsgraph des KNN im Vergleich zu den Trainingsdaten. ©David Söding, 2022

2.3.2 Gradientenverfahren

Durch das Minimieren der Verlustfunktion wird das neuronale Netz optimiert, um die höchstmögliche Performanz zu erreichen. Ein großes Problem, welches im vorherigen Abschnitt nicht beachtet wurde, liegt in der Art, wie das Minimum ermittelt wird. Nur bei wenigen Funktionen ist es möglich, ein Extremum durch eine analytische Lösung wie oben zu finden. Bei vielen Funktionen, unter anderem Exponential-, Logarithmus- und trigonometrische Funktionen, können Extrema nicht durch analytische Methoden berechnet werden, sondern müssen approximativ bestimmt werden. Dies geschieht durch das Gradientenverfahren.

Stellen wir uns vor, wir stehen auf einem nebeligen Berg. Wir können uns lediglich bücken und durch Abtasten die Steigung der jetzigen Position ermitteln. In welche Richtung müssen wir gehen, um am schnellsten und zuverlässigsten möglichst weit abzusteigen?

Eine gute Strategie wäre es, immer in die entgegengesetzte Richtung der Steigung bergab zu gehen. Da das Bücken und Abtasten zeitaufwändig ist, sollten außerdem die Abstände zwischen dem Messen von der Steigung abhängig gemacht werden. Je steiler der Abstieg, desto weiter sollte gegangen werden, bevor erneut gemessen wird. Dies liegt daran, dass der Berg langsam abflacht und eine große Steigung bedeutet, dass man sich noch weit oben befindet. Je flacher die Steigung wird, desto näher ist man an einem Minimum. Da nicht über das Minimum hinweg wieder bergauf gelaufen werden soll, werden die Mess-Abstände bei schwindender Steigung immer kleiner. Auf diese Weise wird garantiert irgendwann einen Punkt mit einer Steigung = 0 erreicht, also einen Tiefpunkt oder Sattelpunkt.

So funktioniert das Gradientenverfahren. Anfangs wird jeder Parameter θ mit einem zufälligen Wert initialisiert. Anschließend wird für jeden Parameter eine partielle Ableitung der Verlustfunktion gebildet und die Steigung relativ zu diesem Parameter an der jetzigen Position berechnet. In einem dreidimensionalen Beispiel würde man sich nach Norden drehen und die Steigung nur in diese Richtung berechnen. Anschließend würde man sich nach Osten drehen und wieder die Steigung nur in Richtung der Ost-West-Achse berechnen. Werden diese beiden Messungen kombiniert, so erhält man den gesamten Gradienten.

Der Gradient der momentanen Position relativ zu dem Parameter mal dem Lerntempo η wird von den Parametern abgezogen. Die Formel dafür ist (verändert aus [4, S. 122]):

$$\theta_i = \theta_i - \eta \frac{\partial}{\partial \theta_i} f(\theta_0, \theta_1, \dots, \theta_n),$$

wobei $\frac{\partial}{\partial \theta_i} f(\cdot)$ die partielle Ableitung der Verlustfunktion nach dem jeweiligen Parameter θ_i ist. Ist der Gradient positiv, so wird der Parameter kleiner, ist er negativ, so wird der Parameter größer. Das sogenannte Lerntempo η beeinflusst die Schrittgröße. Ist η zu gering, so braucht der Algorithmus zu lange, um brauchbare Ergebnisse zu liefern. Ist η zu groß, so springt der Algorithmus unkontrolliert hin und her (siehe Abbildung 12). Der Algorithmus funktioniert, wenn η konstant ist, da bei sinkender Steigung die Schritte automatisch kleiner werden. Allerdings gibt es Optimierungsverfahren wie Adam [8], welche η während des Gradientenverfahrens anpassen, um eine höhere Effizienz zu erzielen. In modernen neuronalen Netzen hat die Verlustfunktion Tausende oder Millionen Variablen, aber durch die partiellen Ableitungen kann trotzdem der Weg bergab gefunden werden.

Probleme können beim Gradientenverfahren auftreten, wenn die Funktion nicht konvex ist, wenn sie also mehrere lokale Minima oder Maxima besitzt. Da durch das Gradientenverfahren nicht klar ist, ob das gefundene Minimum auch das globale Minimum ist, kann das Verfahren in einem lokalen Minimum stecken bleiben und die Verlustfunktion daher nie ausreichend optimiert werden (siehe Abbildung 13) [5,

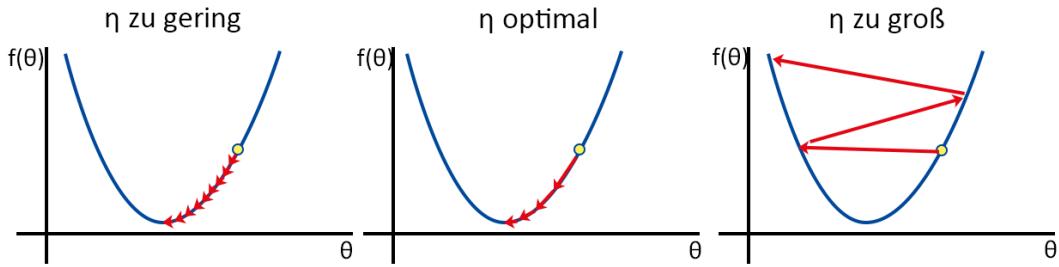


Abb. 12: Vergleich von zu geringem, optimalen und zu hohem Lerntempo. ©David Söding, 2022

S.81]. Oft erreicht das Gradientenverfahren nur ein lokales Minimum. Dies wird aber toleriert, da die Werte der Verlustfunktion für den Anwendungsbereich des KNN niedrig genug sind und weitere Optimierung sich deshalb nicht lohnt [5, S.81]. Ein weiteres Problem sind Plateaus, deren Steigung nahezu Null ist. Durch die gerin-ge Steigung werden die Schritte immer kleiner. Das Plateau wird zwar irgendwann überwunden sein, dies kostet aber viel Zeit und ist sehr ineffizient [4, S.119]. Dieses Problem kann aber durch Momentum Optimierung, wie unter anderem bei Adam, behoben werden [4, S.351].

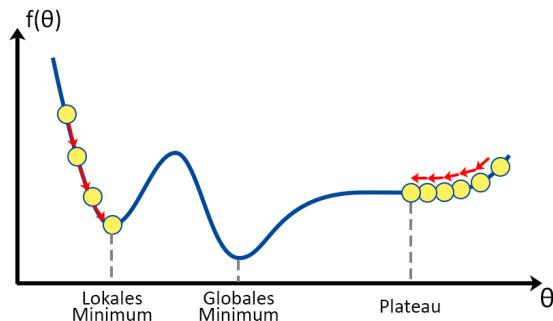


Abb. 13: Problematische Bereiche beim Gradientenverfahren. ©David Söding, 2022

Es gibt verschiedene Varianten des Gradientenverfahrens, die hier nun kurz erläutert werden.

- Das **Batch-Gradientenverfahren** bezieht bei jeder Iteration erneut die gesamten Trainingsdaten in die Berechnung der Gradienten ein. Das Verfahren heißt Batch-Gradientenverfahren, da es bei jedem Trainings-Schritt den gesamten Batch der Trainingsdaten mit einbezieht. Dadurch ist dieses Verfahren bei großen Trainingsdatensätzen sehr langsam [4, 122].

Es ist egal, ob der Gradient der Verlustfunktion des gesamten Batch berechnet wird oder ob mehrere Gradienten jeweils für nur einen Trainingswert berechnet und nachher addiert werden.

- Beim **Stochastik-Gradientenverfahren** wird für jeden Schritt nur eine einzelne zufällige Instanz der Trainingsdaten berücksichtigt, was dieses Verfahren er-

heblich schneller macht. Andererseits ist dieses Verfahren durch die Zufälligkeit wesentlich unregelmäßiger. Die Verlustfunktion sinkt nur im Durchschnitt, einzelne Schritte können auch in die falsche Richtung gehen. Im Gegensatz zu dem Batch-Gradientenverfahren, wird dieses Verfahren zu keinem Minimum konvergieren, sondern in der Nähe des Minimums herumspringen [4, 124].

- Das **Mini-Batch-Gradientenverfahren** ist eine Mischung der beiden vorherigen Verfahren. Statt wie beim Stochastik-Gradientenverfahren nur eine zufällige Instanz zu verwenden, wird beim Mini-Batch-Gradientenverfahren ein zufälliges Bündel (Mini-Batch) mit mehreren Instanzen genutzt. Dadurch ist es zwar langsamer als das Stochastik-Gradientenverfahren, aber dafür regelmäßiger und weniger sprunghaft. Aufgrund dieser Eigenschaften ist es das am weitesten verbreitete der drei Verfahren [4, 127].

2.4 Funktionsweise und Besonderheiten des CNN

Convolutional Neural Networks oder auch CNNs sind eine spezielle Art der neuronalen Netze, welche besonders dafür geeignet sind, Daten, die in einer räumlichen oder zeitlichen Beziehung zueinander stehen, zu verarbeiten. Beispiele hierfür sind Bilder, Mikrofonaufzeichnungen oder Aktienkurszeitreihen. Bei der Bilderkennung beispielsweise sollte nicht jedes Pixel individuell betrachtet werden. Denn besonders die Anordnung der Pixel, also die Position relativ zu den anderen Pixeln, spielt eine sehr wichtige Rolle. Würde jeder Pixel ohne räumlichen Zusammenhang betrachtet, so gingen wichtige Informationen verloren.

Wie ein CNN aufgebaut ist und welche Techniken hinter den einzelnen Bausteinen stecken, wird in den nächsten Abschnitten erklärt.

2.4.1 Convolutional Layer

Der Convolutional Layer ist der wichtigste Bestandteil des CNN. Soll einem CNN als Eingabe ein 32×32 Pixel-Bild geben werden, so hat der Eingabe-Layer 1024 Neuronen. Soll der erste Hidden-Layer nun die gleiche Anzahl an Neuronen besitzen, also eines pro Pixel, so würden $32^2 \cdot 32^2 = 1\,048\,576$ Verbindungen und damit Gewichte als Parameter benötigt, welche alle optimiert werden müssen [2]. Große Mengen an Parametern wirken sich negativ auf die Trainingszeit des CNN aus. Zusätzlich sollen mit dem CNN lokale Muster erkannt werden, weshalb ein einzelnes Neuron keine Verbindung zu allen Pixeln braucht. Stattdessen wird jedes der 32×32 Neuronen jeweils mit dem entsprechenden Pixel plus den 24 benachbarten Pixeln verbunden, also ein 5×5 -Feld an Pixeln. Dadurch wird die Anzahl an Verbindungen auf $32^2 \cdot 5 \cdot 5 = 25\,600$ reduziert [2]. Es lässt sich allerdings noch weiter minimieren, da in jedem Teil des Bildes nach dem gleichen Muster gesucht werden soll. Des-

wegen kann gesagt werden, dass jedes Neuron das gleiche Set an 5×5 -Gewichten auf einem anderen Teil des Bildes anwendet. Dadurch kürzt sich die Anzahl an zu optimierenden Parametern auf $5 \cdot 5 = 25$ herunter. Der Aufbau, der hier beschrieben wurde, lässt sich vereinfacht visualisieren, indem man sich ein 5×5 -Fenster vorstellt, welches in Schritten über das Bild geschoben wird und das Ergebnis jeder Position in einem Neuron speichert [2]. Variieren lässt sich hier die Fenstergröße und die Schrittgröße (siehe Abbildung 14). Wenn das Fenster sich an einem Rand befindet, wird eine Polsterung benötigt, um genug Eingaben zu bekommen. Hierfür können Nullen genutzt werden (siehe Abbildung 14) [2]. Diese 5×5 -Kombination an Gewichten dient als Filter und soll bestimmte Muster im Bild erkennen können. Sie heißen auch Kernel oder Convolution Matrix [5, S.323]. Durch das Training werden die Gewichte des Filters automatisch gelernt.

Wie ein solcher Filter aussehen kann und welche Ergebnisse dadurch entstehen, lässt sich gut bei den von Hand erstellten Kernels in Abbildung 15 erkennen. So lässt sich sogar eine einfache Kantenerkennung durchführen.

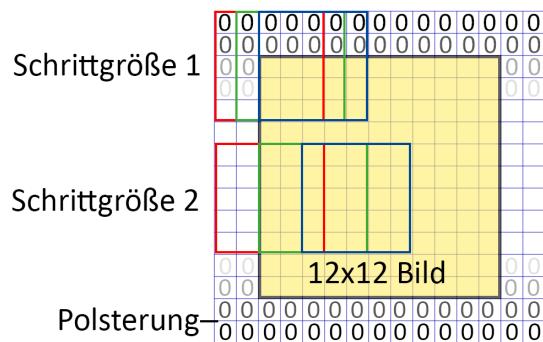


Abb. 14: Polsterung und verschiedene Schrittgrößen bei einem 5×5 Fenster. ©David Söding, 2022

2.4.2 Pooling Layer

Der Hauptzweck eines Pooling Layers ist es, die Komplexität und Anzahl an Neuronen von darauf folgenden Layern zu reduzieren [2]. Zusätzlich sorgt er dafür, dass das CNN von kleinen Änderungen in der Eingabe weniger beeinflussbar ist [5, S.330].

Eine häufige Form des Pooling ist das Max-Pooling. Dabei wird wieder ein Fenster mit variabler Größe und Schrittänge über die Eingaben geschoben und mit jedem Schritt der größte im Fenster enthaltene Wert in einem Neuron ausgegeben. Beim Durchschnitts-Pooling hingegen wird der Mittelwert aller im Fenster enthaltenen Eingaben berechnet (siehe Abbildung 16).

Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Ridge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3×3 (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Abb. 15: Verschiedene Filter und deren Effekt auf ein Foto ©Wikimedia [15].

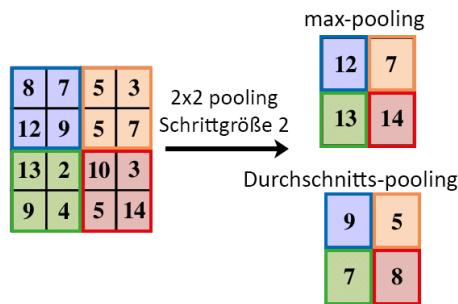


Abb. 16: Beispiele von Max-Pooling und Durchschnitts-Pooling. ©David Söding, 2022

2.4.3 Fully connected Layer

Die letzten Layer eines CNN sind oft fully connected Layer, auch Dense Layer genannt. Sie sind wie normale Layer eines neuronalen Netzes aufgebaut und heißen so, da jedes Neuron in diesem Layer mit jedem Neuron im vorherigen und darauf folgenden Layer verbunden ist [2]. Da Convolutional Layer Muster erkennen, braucht ein CNN zusätzlich normale Layer, die die verschiedenen Muster zu einer einzigen Vorhersage kombinieren. Bei einer typischen Zahlerkennung gibt es 10 Ausgabe-Neuronen, wobei jedes für eine Zahl zwischen 0 und 9 steht. Der letzte Layer gibt an, wie wahrscheinlich es ist, dass die gegebene Eingabe die jeweilige Ausgabe-Zahl ist. Das Neuron mit der höchsten Wahrscheinlichkeit ist der Gewinner und die korrespondierende Zahl somit die Vorhersage des CNN.

3 Entwicklung eines eigenen CNNs in Python

Im folgenden Abschnitt wird erläutert, wie ein Convolutional Neural Network programmiert, trainiert und ausgeführt werden kann. Ziel wird es sein, Bilder der CIFAR-10 Datenbank, einer Kollektion von 60 000 Bildern verschiedener Objekte mit einer Auflösung von 32×32 Pixeln, durch ein Convolutional Neural Network zu kategorisieren. Es gibt zehn Kategorien, nämlich Flugzeuge, Autos, Vögel, Katzen, Rehe, Hunde, Frösche, Pferde, Schiffe und Lastwagen [9]. Die CIFAR-10 Datenbank wird häufig genutzt, um Bilderkennung zu demonstrieren, da die Bilder der Datenbank einheitlich sind, alle die gleiche Auflösung besitzen und bereits kategorisiert sind, sodass nicht mehr jedem Bild die richtige Lösung zugeordnet werden muss. Die Datenbank ist in 50 000 Trainingsbilder und 10 000 Verifikationsbilder eingeteilt. Zusätzlich ist sie randomisiert. Jede Kategorie enthält 6000 Bilder. Kein Bild des Verifikationsteils ist in den Trainingsbildern vorhanden. Außerdem sind alle Objekte aus verschiedene Winkeln und mit unterschiedlichen Hintergründen zu sehen [9]. Dadurch muss das CNN die Variationen lernen und kann sich nicht nur auf eine Version pro Objekt-Klasse einstellen.

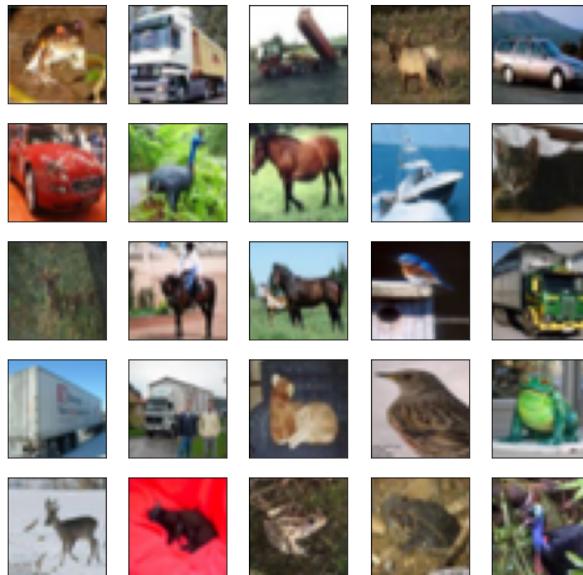


Abb. 17: 25 Beispielbilder der CIFAR-10 Datenbank

Im folgenden Abschnitt wird ein mithilfe der Keras-API-Referenz ([7]) eigenständig erstellter Code erläutert. Kein Teil des Codes wurde direkt übernommen. Die Keras-API-Referenz dient lediglich als Wörterbuch und Erklärung für Klassen und deren Methoden.

3.1 Implementierung

In diesem Abschnitt wird zuerst die Installation aller nötigen Programme und Bibliotheken behandelt. Danach wird im Detail auf das Konstruieren und Programmieren

des Modells eingegangen.

3.1.1 Installation und Setup

Zu allererst werden ein paar Dinge installiert. Auf der offiziellen [Website](#) von Python lässt sich die neuste Version der Programmiersprache herunterladen. Hierbei muss darauf geachtet werden, die richtige Version für das eigene Betriebssystem auszuwählen. Anschließend führt man den Installer aus und folgt den Anweisungen. Gelangt man zu den Optionen **Install Now** und **Customize Installation**, so wird Letzteres gewählt. Im folgenden Feld können alle Häkchen gesetzt werden. Besonders wichtig ist hier **pip**. Pip ist ein Package-Manager, der die Installation von Bibliotheken vereinfacht. Anschließend wird den Installationsanweisungen bis zum Schluss gefolgt.

Nachdem das Setup beendet ist, kann mit Hilfe von pip die Tensorflow Bibliothek installiert werden. Hierfür wird die Eingabeaufforderung geöffnet. In Windows 10 lässt sich dafür die Windows-Taste drücken und `cmd` eingeben. Mit Enter öffnet sich dann das Eingabefenster. Dort wird der Befehl

```
py -m pip install tensorflow
```

eingegeben [12]. Nach einer kurzen Wartezeit ist Tensorflow installiert. Tensorflow ist ein Machine-Learning-System von Google, welches uns viele Aufgaben abnimmt [1]. Es muss also kein neuronales Netz aus dem Nichts programmiert werden. Stattdessen werden die Bausteine und Methoden, die Tensorflow zur Verfügung stellt, direkt genutzt.

Da Tensorflow alleine sehr komplex ist, wird Keras genutzt, eine Python-Bibliothek, um nicht direkt mit Tensorflow und den damit verbundenen mathematischen Details umgehen zu müssen. Während Tensorflow als Backend agiert, ist Keras das Frontend und schafft somit eine Schnittstelle, mit der es sich leichter arbeiten lässt [11]. Keras wird mit Hilfe des folgenden Befehls installiert [12]:

```
py -m pip install keras
```

Zusätzlich wird mit dem Befehl

```
py -m pip install matplotlib
```

eine Bibliothek installiert, die es ermöglicht, Graphen und Bilder zu generieren.

Anschließend wird nur noch eine Benutzeroberfläche zur einfachen und effizienten Bedienung benötigt. Eine sehr einfach zu verwendende Oberfläche ist Jupyter. Diese wird installiert durch den Befehl

```
py -m pip install jupyter
```

Um Jupyter zu starten, wird der Befehl

```
jupyter notebook
```

eingegeben, wodurch der Browser mit dem Link `http://localhost:8888/tree` geöffnet wird [5, S.44]. Sollte dies nicht geschehen, so lässt sich der Link auch manuell in der URL eingeben. In diesem Jupyter Interface erstellt man ein neues Notebook, also ein neues Python-Projekt (siehe Abbildung 18).

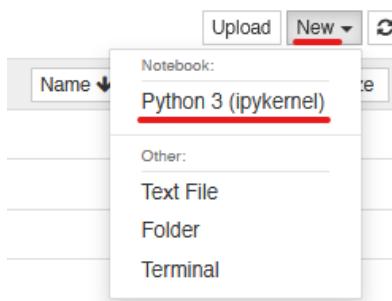


Abb. 18: Klicke auf New und anschließend Python 3.

3.1.2 Erstellen des Modells

Mithilfe dieser Befehle

```
import tensorflow as tefl
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

werden die benötigten Bibliotheken in das Projekt importiert. In Jupyter wird der Befehl in das Textfeld geschrieben und anschließend auf den Knopf **Run** geklickt, um die Befehle auszuführen.

Mit der `datasets` Bibliothek von Keras lässt sich nun über diesen Befehl die CIFAR-10 Datenbank importieren:

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
```

In den Reihungen `train_images` und `test_images` sind die Bilddateien gespeichert, während in `train_labels` und `test_labels` die Klassifikationen, also die Lösungen, gespeichert sind.

Um zu überprüfen, ob die CIFAR-10 Datenbank richtig importiert wurde und einen Eindruck von dem Datensatz zu bekommen, kann mit Hilfe der `matplotlib`-Bibliothek eine einfache Grafik erstellt werden. Diese Grafik beinhaltet die ersten 25 Bilder (siehe Abbildung 17).

```
plt.figure(figsize=(8,8))      #Abbildung wird erstellt und Groesse festgelegt
for i in range(25):            #for-Schleife, die 25 mal ausgefuehrt wird
    plt.subplot(5,5,i+1)        #Bild wird in einem 5x5 Gitter angeordnet
```

```

plt.axis("off")           #Achsen-Beschriftung wird deaktiviert
plt.imshow(train_images[i]) #i'tes Bild des Trainingssets wird angezeigt

```

Da die Pixel-Werte von 0 bis 255 reichen, das Netz aber Eingaben zwischen 0 und 1 bekommen soll, werden mit diesem Befehl alle Pixel-Werte durch 255 geteilt:

```

train_images = train_images / 255
test_images = test_images / 255

```

Nun kann begonnen werden, das Modell für ein Convolutional Neural Network zu erstellen. Diese gut funktionierende Anordnung von Layern wurde (von mir) durch Ausprobieren gefunden. Es scheint sinnvoll zu sein, Convolutional und Pooling Layer abzuwechseln.

```

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), strides=(1, 1), activation='relu',
                      padding='same', input_shape=(32, 32, 3)))
model.add(layers.Conv2D(32, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Conv2D(64, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Conv2D(64, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Conv2D(128, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

```

Die erste Zeile definiert die Art des Modells. Hier wird eine lineare Aneinanderreihung verschiedener Layer benutzt. Die erste Zahl nach den Klammern bei den Convolutional Layern sind die Anzahl an Filtern, also wie viele Fenster in einem Layer über die Eingaben geschoben werden. Es ist sinnvoll, die Anzahl an Filtern bei den tieferen Layern zu erhöhen, da anfangs nur einfache Muster wie Kanten erkannt werden und die Muster bei tieferen Layern komplizierter werden, weshalb mehr Filter nötig sind, um sie alle zu erkennen. Die zweite und dritte Zahl bestimmen die Fenstergröße, welche bei allen Convolutional Layern die Dimensionen 3×3 hat. `strides=(1, 1)` bedeutet, dass die Schrittgröße des Fensters in x- sowie y-Richtung eins ist. Die Aktivierungsfunktion ist bei allen Convolutional Layern ReLU, während `padding='same'` eine Polsterung aus Nullen aktiviert. Beim ersten Layer müssen zusätzlich die Dimensionen der Eingaben eingegeben werden. Da die Bilder der Datenbank eine Auflösung von 32×32 haben und zusätzlich drei Farb-Kanäle besitzen (Rot, Grün und Blau), ist die Auflösung der Eingaben $32 \times 32 \times 3$. Zusätzlich zu den Convolutional Layern gibt es noch 3 Max-Pooling Layer mit einer Fenstergröße und Schrittgröße von 2×2 [7].

Nun kann mit Hilfe des Befehls

```
| model.summary()
```

das bisher konstruierte Netz betrachtet werden. Als Ausgabe erhalten wir Folgendes:

```
Model: "sequential"
Model: "sequential"
-----
Layer (type)          Output Shape       Param #
=====
conv2d (Conv2D)        (None, 32, 32, 32)    896
conv2d_1 (Conv2D)      (None, 32, 32, 32)    9248
max_pooling2d (MaxPooling2D) (None, 16, 16, 32) 0
)
conv2d_2 (Conv2D)      (None, 16, 16, 64)    18496
max_pooling2d_1 (MaxPooling2D) (None, 8, 8, 64) 0
)
conv2d_3 (Conv2D)      (None, 8, 8, 64)    36928
max_pooling2d_2 (MaxPooling2D) (None, 4, 4, 64) 0
)
conv2d_4 (Conv2D)      (None, 4, 4, 128)   73856
max_pooling2d_3 (MaxPooling2D) (None, 2, 2, 128) 0
)
=====
Total params: 139,424
Trainable params: 139,424
Non-trainable params: 0
```

Der erste Teil des Modells ist nun konstruiert, aber wie sich erkennen lässt, sind die Dimensionen der Ausgabe des letzten Layers $4 \times 4 \times 128$ (anstelle von None steht später die Größe des Batch, welche variabel ist und daher nicht im Vorhinein festgelegt werden kann[7]). Die Ausgabe des letzten Layers des Netzes darf allerdings nur eine Dimension haben und aus 10 Neuronen bestehen, da der CIFAR-10-Datensatz in 10 Kategorien unterteilt werden soll. Um dies zu erreichen, wird dem Modell noch ein zweiter Teil angefügt.

```
| model.add(layers.Flatten())
| model.add(layers.Dense(64, activation='relu'))
| model.add(layers.Dense(10, activation='softmax'))
```

Der erste Layer komprimiert die Ausgabe auf eine Dimension. Der zweite Layer ist ein fully connected Layer mit 64 Neuronen und der ReLU als Aktivierungsfunktion. Der letzte fully connected Layer hat nun 10 Neuronen und dient als Ausgabe-Layer. Die Aktivierungsfunktion Softmax wandelt einen Vektor mit Werten in eine Wahrscheinlichkeitsverteilung um. Die Werte der Ausgabe ergeben summiert eins [7]. Diese Funktion wird benötigt, da das CNN am Ende für das Bild die Kategorie angeben soll, der es die höchste Wahrscheinlichkeit zuteilt.

Wird nun noch einmal der Befehl `model.summary()` ausgeführt, so sehen wir, dass das Convolutional Neural Network jetzt 172 906 Parameter besitzt. Um diese nun zu optimieren, muss das eben erstellte Modell kompiliert werden.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
              metrics=['accuracy'])
```

Die Methode `model.compile` bereitet das Modell auf das Training vor. Ein paar Einstellungen können hier noch eingegeben werden, wie zum Beispiel das Optimierungsverfahren. In diesem Fall wird der zuvor erwähnte Optimizer Adam genutzt. Dieser passt das Lerntempo η basierend auf den vorherigen Gradienten-Schritten so an, dass das Ziel schneller erreicht wird [8]. Zusätzlich wird die Verlustfunktion festgelegt. Für diesen Fall wird `SparseCategoricalCrossentropy` genutzt, da diese Verlustfunktion für das Einteilen in Kategorien gedacht ist. Sie bewertet nicht nur, ob die Vorhersage richtig ist, sondern auch, wie zutreffend die Wahrscheinlichkeitsverteilung ist. Es muss hier zusätzlich noch `from_logits=False` spezifiziert werden, da die Ausgabe des Netzes eine Wahrscheinlichkeitsverteilung ist [7]. Als letztes wird der Methode noch angegeben, dass die Metrik `accuracy`, also die Genauigkeit, während des Trainings ausgewertet werden soll.

Da das CNN nun kompiliert ist, kann es trainiert werden. Hierfür wird die `fit`-Methode genutzt:

```
statistics = model.fit(train_images,
                       train_labels,
                       batch_size=64,
                       steps_per_epoch=782,
                       epochs=10,
                       validation_data=(test_images, test_labels))
```

Als Parameter werden hier die Trainingsbilder und die korrekten Trainings-Kategorien (`train_labels`) übergeben, also die Lösungen. Da hier das Mini-Batch Gradientenverfahren genutzt wird, muss spezifiziert werden, wie groß der Mini-Batch sein soll. Nach einigen Experimenten ergab sich ein Wert von 32 oder 64 als zielführend. Es sollte hier eine Zweierpotenz gewählt werden, um die Rechenleistung und Speicherkapazität der CPU voll auszunutzen. Die `steps_per_epoch` ist die Anzahl der Schritte des Gradientenverfahrens pro Epoche. Da in jeder Epoche oder auch Generation einmal die gesamten Trainingsdaten verwendet werden sollen, wird für die Schritte pro Epoche die Anzahl an Trainingsbildern geteilt durch die Batch-Größe gewählt. $50000/64$ ist 781.25, also runden wir auf 782 auf. Jede Epoche stellt also einen Durchlauf des Trainingsdatensatzes dar. Nach jeder Epoche wird die Genauigkeit des CNN anhand der Verifikationsdaten (`validation_data`) gemessen. Diese Daten werden nicht für das Training verwendet, wodurch sich das neuronale Netz diese nicht merkt. Es wurden hier 10 Epochen gewählt, da das Training so nicht all-

zu lange dauert und trotzdem gute Ergebnisse erwarten werden können. Die Daten, die während des Trainings aufgenommen werden, werden in dem Objekt `statistics` gespeichert. Diese können genutzt werden, um das Training des CNNs genau zu analysieren.

Während das Netzwerk trainiert wird, werden nach jeder Epoche ein paar Daten ausgegeben. Für die zehnte Epoche sieht dies so aus:

```
Epoch 10/10
782/782 [=====] - 146s 187ms/step - loss: 0.3111 -
accuracy: 0.8905 - val_loss: 0.8314 - val_accuracy: 0.7611
```

`loss` und `val_loss` sind die Werte der Verlustfunktion für die Trainings- und Verifikationsdaten. Es ist zu erwarten, dass `val_loss` größer als `loss` ist, da sich das CNN auf die Trainingsdaten angepasst hat. `accuracy` und `val_accuracy` beschreiben die Prozentzahl an richtig kategorisierten Bildern. Auch `accuracy` ist höher als `val_accuracy`, da das CNN auf die Trainingsdaten abgestimmt ist, aber nie mit den Verifikationdaten trainiert wurde. Mit dem Befehl:

```
| print(model.evaluate(test_images, test_labels))
```

lässt sich die finale `val_accuracy` herausfinden. Dieses Modell hat nach zehn Epochen eine Genauigkeit von 76.11% erreicht.

Hiermit lässt sich ein Graph von `accuracy` und `val_accuracy` plotten (siehe Abbildung 19):

```
plt.plot(statistics.history['accuracy'], label='accuracy')
plt.plot(statistics.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
```

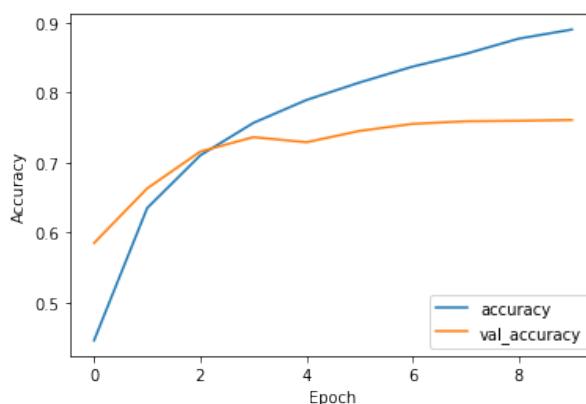


Abb. 19: `accuracy` und `val_accuracy` im Verlauf des Trainings. ©David Söding, 2022

Für die Performanz des CNN ist hauptsächlich die Genauigkeit auf den Verifikationsdaten zu betrachten. Es lässt sich in Abbildung 19 erkennen, dass das Maximum schon fast ab Epoche 6 erreicht wurde. Daraus lässt sich schließen, dass weiteres

Training dem CNN nicht verbessert hätte und es die maximale durch Training erreichbare Genauigkeit erreicht hat. Diese Vermutung lässt sich mit Abbildung 20 bestätigen. Es lässt sich sogar erkennen, dass die Genauigkeit etwas sinkt, was an Überanpassung (englisch overfitting) liegen kann: Wenn ein Netzwerk zu viel trainiert wird, kann es passieren, dass es sich zu sehr an Details der Trainingsdaten anpasst und dadurch schlechter verallgemeinern kann und somit schlechter auf neuen Daten funktioniert. Da der Trainingsdatensatz sehr groß ist, ist der Effekt des Übertrainierens hier nur sehr klein [5, S. 239].

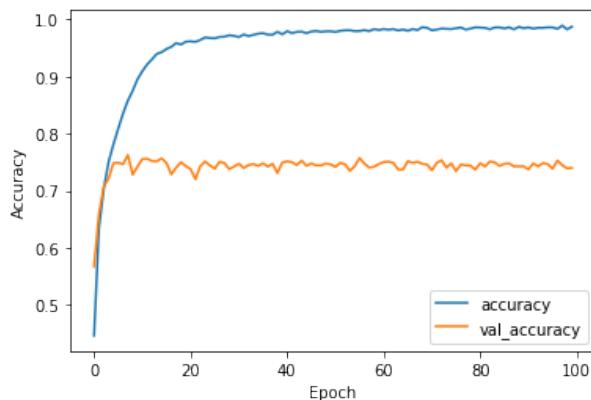


Abb. 20: accuracy und val_accuracy im Verlauf des Trainings bis zu Epoche 100.
©David Söding, 2022

3.1.3 Weitere Verbesserungen

Um dieses Netzwerk nun weiter zu verbessern, kann eine weitere Layer-Art eingeführt werden. Sogenannte Dropout-Layer setzen zufällig Neuronen in diesem Layer auf Null [7], löschen diese also für einen Gradienten-Schritt, also ein Mini-Batch, aus dem Netzwerk. Dadurch wird verhindert, dass das neuronale Netz sich zu sehr an die Trainingsdaten anpasst, da sich das neuronale Netz jetzt nicht mehr vollständig auf einzelne Neuronen stützen kann. Stattdessen ist es gezwungen, zu verallgemeinern, was in einer höheren Verifikations-Genauigkeit resultiert [5, S.251]. Anfangs wird das Netzwerk schlechtere Ergebnisse erzielen, aber mit der Zeit lässt sich erkennen, wie effektiv Dropout-Layer sind.

Es wurden drei verschiedene Konfigurationen für Dropout-Layer für jeweils 100 Epochen getestet. Die drei Modelle haben folgenden Aufbau:

1.	2.	3.
Conv2D	Conv2D	Conv2D
Conv2D	Conv2D	Conv2D
MaxPooling2D	MaxPooling2D	MaxPooling2D
Dropout(0.2)	Dropout(0.1)	Dropout(0.2)
Conv2D	Conv2D	Conv2D
MaxPooling2D	MaxPooling2D	MaxPooling2D

Dropout(0.3)	Conv2D	Dropout(0.3)
Conv2D	MaxPooling2D	Conv2D
MaxPooling2D	Dropout(0.2)	MaxPooling2D
Dropout(0.4)	Conv2D	Dropout(0.4)
Conv2D	MaxPooling2D	Conv2D
MaxPooling2D	Dropout(0.3)	MaxPooling2D
Dropout(0.5)	Flatten	Dropout(0.5)
Flatten	Dropout(0.4)	Flatten
Dense(64)	Dense(128)	Dense(64)
Dense(10)	Dropout(0.4)	Dropout(0.5)
	Dense(64)	Dense(10)
	Dropout(0.4)	
	Dense(10)	
.		
val_accuracy nach 100 Epochen:		
0.8220	0.7995	0.8050

Da Modell 1 nach 100 Epochen die beste Genauigkeit hat (siehe Abbildung 21) wird nun mit diesem Modell fortgefahrene.

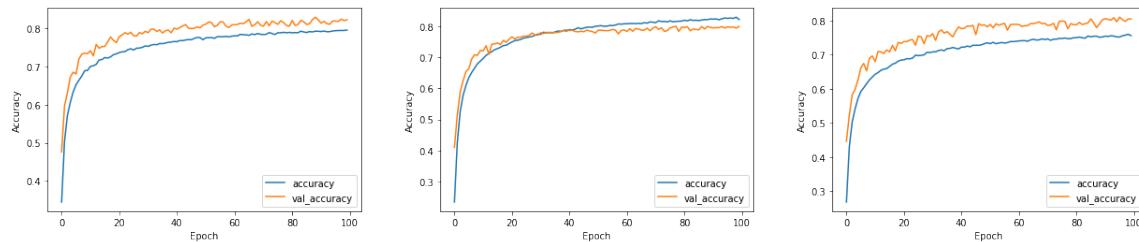


Abb. 21: Von links nach rechts: val_accuracy von Modell 1, Modell 2 und Modell 3.
©David Söding, 2022

Um das neue Modell zu erstellen, wird folgender Code genutzt (Parameter wie stride etc. hier der Übersicht halber nicht aufgeschrieben):

```
model = models.Sequential()
model.add(layers.Conv2D())
model.add(layers.Conv2D())
model.add(layers.MaxPooling2D())
model.add(layers.Dropout(0.2))
model.add(layers.Conv2D())
model.add(layers.MaxPooling2D())
model.add(layers.Dropout(0.3))
model.add(layers.Conv2D())
model.add(layers.MaxPooling2D())
model.add(layers.Dropout(0.4))
model.add(layers.Conv2D())
model.add(layers.MaxPooling2D())
model.add(layers.Dropout(0.5))

model.add(layers.Flatten())
```

```

model.add(layers.Dense(64))
model.add(layers.Dense(10))

```

Die Zahl hinter dem Dropout-Layer gibt an, wie viele Neuronen betroffen sind. 1 bedeutet, dass alle betroffen sind, und 0, dass keines betroffen ist. Wie sich erkennen lässt, erhöht sich die Rate in den späteren Layern. Dies liegt daran, dass bei der steigenden Komplexität von Mustern mehr verallgemeinert werden soll, wohingegen im ersten Layer eine einfache Kantenerkennung beispielsweise nicht verallgemeinert werden muss.

Nach 100 Epochen des Trainierens ergab sich mit diesem Modell eine `val_accuracy` von 82.20% (siehe Abbildung 22). Dass `val_accuracy` hier konstant höher ist als `accuracy`, liegt daran, dass der Dropout-Layer ausschließlich beim Trainieren aktiv ist. Für die tatsächliche Bilderkennung ist dieser nur hinderlich, weshalb er nicht auf die Verifikationsbilder angewandt wird.

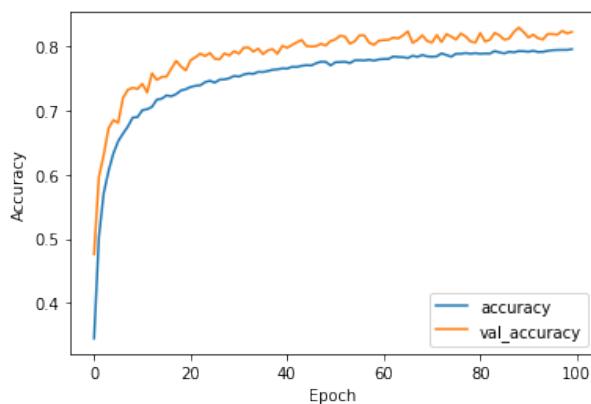


Abb. 22: `accuracy` und `val_accuracy` mit Dropout im Verlauf des Trainings bis zu Epoche 100. ©David Söding, 2022

Das letzte, was zusätzlich getestet werden kann, ist die Layer- und Batch-Normalisierung. Durch die Gradientenberechnung kann es passieren, dass die Gradienten der Parameter bei späteren Layern immer kleiner werden. Dadurch werden die Parameter der ersten Layer kaum verändert, wodurch das Training behindert wird. Im Gegensatz dazu kann auch das Gegenteil auftreten. Je tiefer sich die Layer im Netz befinden, desto größer werden die Gradienten, wodurch die Parameter viel zu stark verändert werden. Dieses Phänomen nennt sich Vanishing/Exploding Gradients [4, S.332].

Normalisierungs-Layer können diesem Problem entgegenwirken. Sie normalisieren die Werte des Netzes, indem sie sie mit dem Mittelwert der zu normalisierenden Werte subtrahieren, somit um Null zentrieren, und sie durch die Standardabweichung der Werte teilen [7]. Es gibt in Keras zwei verschiedene Normalisierungs-Layer. Die Batch-Normalisierung normalisiert alle Werte eines Neurons über den gesamten Batch. Bei einer Batch-Größe von 64 hat ein einzelnes Neuron also 64 verschiedene Werte, welche relativ zueinander normalisiert werden. Die Layer-Normalisierung normalisiert alle Neuronen eines Layers für nur eine Einheit des Batch [7]. Wel-

che dieser Methoden für unser Netz besser funktioniert, lässt sich am besten durch Ausprobieren ermitteln.

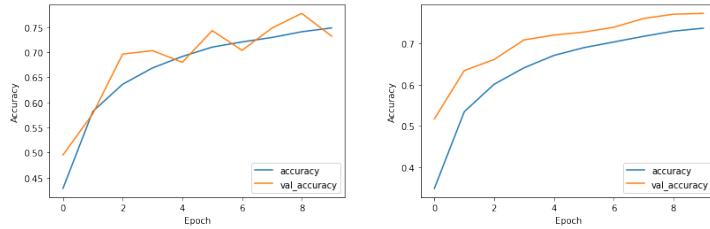


Abb. 23: Links: Batch-Normalisierung, Rechts: Layer-Normalisierung nach zehn Epochen. ©David Söding, 2022

Wie sich in Abbildung 23 erkennen lässt, scheinen beide Normalisierungs-Arten zu funktionieren. Ohne Normalisierung hatte das CNN nach zehn Epochen eine val_accuracy von 74%. Mit Batch-Normalisierung erreicht das Netz in Epoche 9 schon eine val_accuracy von 77%. Die Werte schwanken aber stark, wodurch es in Epoche 10 nur noch 73% sind. Die Werte des Netzes mit Layer-Normalisierung sehen wesentlich einheitlicher aus. Es hat eine Genauigkeit von 77% in der zehnten Epoche erreicht. Es könnte sein, dass die Batch-Normalisierung eventuell eine größere Verbesserung bewirken kann, allerdings scheint die weniger schwankende Layer-Normalisierung eine sicherere und daher bessere Wahl zu sein. Fügt man nun nach jedem Convolutional-Layer und ein Mal als vorletzten Layer einen Layer-Normalisierungs-Layer ein, so ergibt sich folgender Code:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), strides=(1, 1), activation='relu',
                      padding='same', input_shape=(32, 32, 3)))
model.add(layers.LayerNormalization())
model.add(layers.Conv2D(32, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.LayerNormalization())
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Dropout(0.2))
model.add(layers.Conv2D(64, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.LayerNormalization())
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Dropout(0.3))
model.add(layers.Conv2D(64, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.LayerNormalization())
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Dropout(0.4))
model.add(layers.Conv2D(128, (3, 3), strides=(1, 1), activation='relu',
                      padding='same'))
model.add(layers.LayerNormalization())
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
```

```

model.add(layers.Dropout(0.5))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.LayerNormalization())
model.add(layers.Dense(10, activation='softmax'))

```

Nachdem dieses Modell für 100 Epochen trainiert wurde, ergibt sich der Graph aus Abbildung 24. Das Modell erreicht mehrmals eine `val_accuracy` von über 85%, schwankt aber gegen Ende leicht, wodurch es nach Epoche 100 eine Genauigkeit von 84.88% hat. Nicht nur hat sich die Geschwindigkeit des Trainings deutlich erhöht, das Netz hat durch die Normalisierung zusätzlich drei Prozentpunkte an Genauigkeit dazugewonnen.

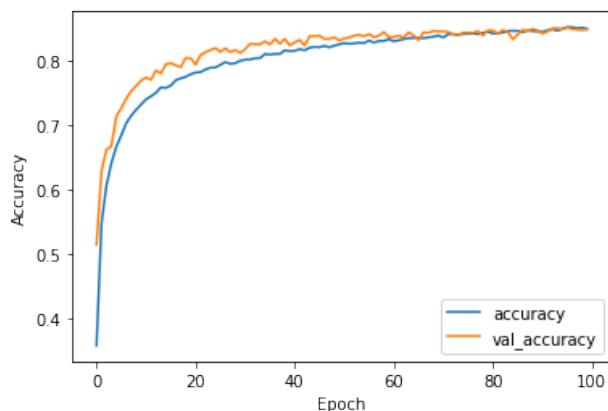


Abb. 24: Modell mit Layer-Normalisierung nach 100 Epochen. ©David Söding, 2022

3.2 Evaluierung des CNN

Da das Netz nun fertig trainiert ist, kann es mit selbstgewählten Bildern getestet werden. Um Bilderkennung in echten Anwendungsbereichen bestmöglich zu simulieren, wurden keine Bilder aus einem Datensatz verwendet. Stattdessen wurden Stock-Fotos von Websites wie [Pexels](#) heruntergeladen. Um die Bilderkennung zu vereinfachen, wurden manuell die Objekte in den Bildern zentriert und die Auflösung auf $32 \cdot 32$ Pixel reduziert (siehe Abbildung 25). Hierfür könnte natürlich auch ein Programm geschrieben werden, welches diese Aufgabe automatisch übernimmt.

Die Testbilder können nun über den Upload-Knopf im Jupyter Homescreen in Jupyter importiert werden. Anschließend werden zwei weitere Bibliotheken benötigt, um die Bilder nutzen zu können.

```

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

```

Anschließend müssen die Bilder in das Programm importiert und umgewandelt werden:

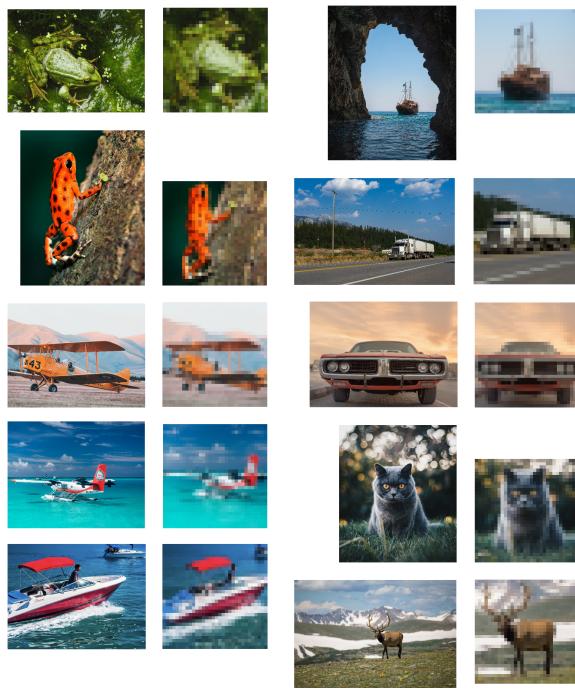


Abb. 25: Original-Bilder und angepasste Bilder zum Testen des CNN.

```
#Bilder werden geladen und in img Objekt gespeichert.
img = (load_img("frog32.png"), load_img("2frog32.png"),
       load_img("airplane32.png"), load_img("2airplane32.png"),
       load_img("2ship32.png"), load_img("ship32.png"),
       load_img("truck32.png"), load_img("car32.png"),
       load_img("cat32.png"), load_img("deer32.png"))

#Bilddaten werden in dreidimensionale Arrays umgewandelt (32x32x3).
img = (img_to_array(img[0]), img_to_array(img[1]),
       img_to_array(img[2]), img_to_array(img[3]),
       img_to_array(img[4]), img_to_array(img[5]),
       img_to_array(img[6]), img_to_array(img[7]),
       img_to_array(img[8]), img_to_array(img[9]))

#Einzelne Arrays werden zu einem Array mit den Dimensionen 32x32x3x10
#zusammengefügt. Die vierte Dimension ist die Anzahl an Test-Bildern.
img = tf.stack(img)

#Pixel-Werte werden von 0 bis 255 auf 0 bis 1 eingeschraenkt.
img = img / 255.0
```

Da die Bilder nun importiert und richtig formatiert sind, kann das CNN beginnen die Objekte zu erkennen. Hierfür wird der Befehl `model.predict(img)` genutzt:

```
#Da das CNN als Ergebnis nur Zahlen ausgeben kann, müssen diese in Wörter umgewandelt
#werden. Dafür wird dieser Array genutzt. Jeder Index steht für ein Wort.
names = ['Flugzeug', 'Auto', 'Vogel', 'Katze', 'Reh',
         'Hund', 'Frosch', 'Pferd', 'Schiff', 'Lastwagen']

#Bilder werden an predict-Methode übergeben und für jedes Bild wird der Index
#der höchsten Wahrscheinlichkeit in result abgespeichert.
result = (tf.math.argmax(model.predict(img), axis=1))

#Alle Wahrscheinlichkeiten werden in percents abgespeichert.
```

```

percents = model.predict(img)
#Index der Vorhersage (result[0]) wird mit Hilfe des Wort-Arrays in Wort umgewandelt
#und ausgegeben.
print(names[result[0]])
#Prozentzahl der Vorhersage wird ausgegeben.
print(percents[0][result[0]])
#Ergebnisse werden ausgegeben.
print(names[result[1]])
print(percents[1][result[1]])

print(names[result[2]])
print(percents[2][result[2]])

...
#3 bis 8 wurde hier ausgeblendet.

print(names[result[9]])
print(percents[9][result[9]])

```

Nun gibt das Programm Folgendes aus:

Frosch	Schiff
0.9967817	0.9998994
Frosch	Lastwagen
0.91471094	0.99290985
Flugzeug	Auto
0.94807357	0.871465
Flugzeug	Katze
0.9979584	0.5982983
Schiff	Reh
0.995952	0.98688394

Die Zahlen nach den Wörtern geben an, wie stark die Wahrscheinlichkeitsverteilung um die Kategorie konzentriert ist. Je näher die Zahl an der Eins ist, desto konzentrierter ist die Verteilung. Die Zahlen können als vermutete Wahrscheinlichkeit interpretiert werden. Hier wurden für das Programm schwierig zu erkennende Bilder gewählt, wie beispielsweise die zwei verschiedenfarbigen Frösche, das Wasserflugzeug und das gleichfarbige Schnellboot, sowie das Auto aus einer ungewöhnlichen Front-Perspektive (siehe Abbildung 25). Trotzdem hat das neuronale Netz jedes Bild richtig erkannt. Interessanterweise ist die Verteilung bei der Katze am flachsten, obwohl das Bild leicht erkennbar zu sein scheint.

Um das Netz an seine Grenzen zu bringen, wurde noch etwas mehr ausgetestet (siehe Abbildung 26). Das Bild einer Eule wurde mit einer 88% Wahrscheinlichkeit als Vogel eingestuft, obwohl eine Eule ein sehr untypischer Vogel ist und vermutlich nicht oft im Training vertreten ist. Das Pferd wurde mit einer Wahrscheinlichkeit von 54% als Frosch eingestuft, während das Bild eines Fahrrades, welches nicht einmal eine Kategorie ist, mit 64% als Reh eingestuft wurde. Ein Bild mit völlig zu-

fälligen Pixel-Werten (siehe Abbildung 26) wurde mit einer Wahrscheinlichkeit von 99% als Lastwagen eingestuft. Es lässt sich hier also deutlich erkennen, dass das Netz die Wahrscheinlichkeitsverteilung, auch wenn die Vorhersage falsch ist, relativ eindeutig um eine der Kategorien konzentriert. Dies könnte an der Verlustfunktion liegen, die im Training genutzt wird, da diese nicht nur bewertet, ob die Kategorie mit der höchsten Wahrscheinlichkeit richtig ist, sondern auch die Abweichung der Wahrscheinlichkeiten der anderen Werte beachtet. Also ist es für das CNN sinnvoller, eine stark konzentrierte Verteilung auszugeben, als eine flache, auch wenn die Vorhersage falsch ist.

Zusätzlich wurde getestet, ob das CNN die Objekte auch am Hintergrund erkennt. Tiere sind oft vor einem grünen Hintergrund, Schiffe und Flugzeuge vor etwas Blauem. Deshalb wurde ein Bild eines Schiffes ohne Hintergrund getestet (siehe Abbildung 26). Auch dieses Bild wurde von dem neuronalen Netz erkannt, und dies mit einer Wahrscheinlichkeit von 99%. Eine Vermutung war, dass sich das CNN eventuell an der typisch roten Farbe des Schiffes orientiert. Also wurde das Bild so bearbeitet, dass das Schiff eine oliv-grüne Farbe hat (siehe Abbildung 26). Aber auch dieses Bild wurde mit einer Wahrscheinlichkeit von 99% als Schiff eingestuft. Dies lässt vermuten, dass das CNN neben Farbe und Hintergrund auch die tatsächlichen Formen und Muster der Objekte gelernt hat.

Es wurde auch getestet, ein Flugzeug und Auto selbst zu zeichnen und dabei typische Formen und Farben beizubehalten (siehe Abbildung 26). Trotz der starken Abstrahierungen konnte das CNN beide Bilder mit über 95% Wahrscheinlichkeit richtig einordnen.

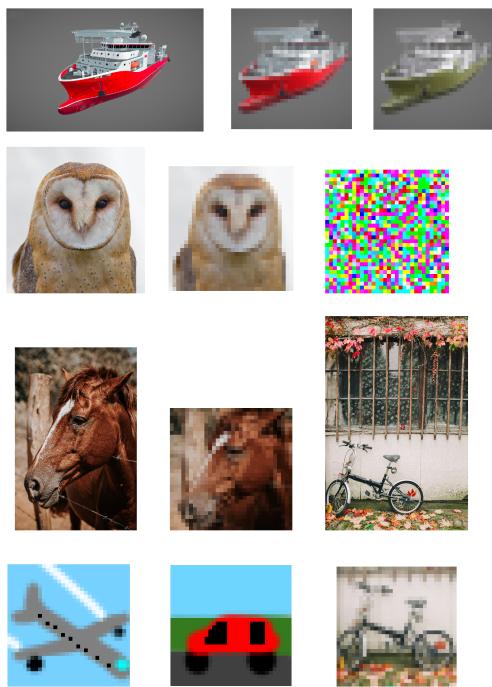


Abb. 26: Schwierige Bilder zum Testen des CNN.

4 Abschließende Bemerkungen

Die gewonnenen Erkenntnisse offenbaren, dass das selbst modellierte und trainierte Convolutional Neural Network über bloße Farberkennung hinaus in der Lage ist, komplizierte Formen, Muster und Kombinationen von Mustern zu erkennen. Es ist zwar weit davon entfernt, perfekt zu sein, für die einfache Struktur und geringe Trainingsdauer ist es aber überraschend leistungsstark.

Was allerdings verstanden werden sollte, ist dass neuronale Netze trotz der beachtlichen Fähigkeiten fundamental anders funktionieren als menschliche Gehirne. Die meisten Machine-Learning-Techniken wurden entwickelt, um sehr spezifische Probleme lösen zu können. Sie sind in ihren Fähigkeiten sehr eingeschränkt und können nur exakt solche Probleme lösen, auf die sie trainiert wurden. Wurden in den Trainingsdaten beispielsweise Fotos, welche in der Nacht aufgenommen wurden, weggelassen, so kann es sein, dass die Bilderkennungs-Software die Hälfte des Tages nutzlos ist. Jedes neuronale Netz ist auf die Trainingsdaten angewiesen und wird in diesen Trainingsdaten etwas übersehen, so kann dieses im echten Anwendungsbereich starke Folgen haben. Dieses Problem offenbart die sogenannte Adversarial Attack (feindlicher Angriff). Das Ziel dieses Angriffes ist es, bewusst Inputs zu manipulieren, damit diese von dem Netz falsch klassifiziert werden, während Menschen die Manipulation nicht auffällt oder nicht verdächtig vorkommt. Dadurch kann in Zukunft besser gegen derartige Situationen vorgegangen werden und neuronale Netze zusätzlich besser verstanden werden. Eine der bekanntesten Anwendungsbereiche der Adversarial Attack ist die Verkehrszeichen-Klassifizierung. Forschern ist es gelungen, Verkehrszeichen wie beispielsweise ein Stopp-Schild minimal abzukleben, so dass ein Klassifizierungs-Algorithmus, wie er auch in einem echten Auto vorkommen könnte, dieses Schild nicht mehr als Stoppschild, sondern als Geschwindigkeitsbegrenzung 45 einordnet (siehe Abbildung 27) [3]. Adversarial Attacks lassen sich auf fast alle Gebiete des Machine-Learning anwenden. Oft sind sie für das menschliche Auge nicht sichtbar, da lediglich ein bestimmtes Muster mit einem sehr geringem Gewichtsfaktor auf die Daten angewandt wird (siehe Abbildung 28) [6]. So könnten eventuell auch Spam- und Medien-Filter umgangen werden.

Der Bereich der künstlichen Intelligenz wächst stetig. Es wird in allen Bereichen der Wissenschaft immer wichtiger, Kenntnisse in diesem Bereich zu besitzen, und die Fähigkeiten der neuronalen Netze nehmen zu. Einerseits sind Machine-Learning-Modelle in speziellen Aufgaben schon längst besser als medizinische Experten, andererseits fängt in vielen Bereichen die Nutzung solcher Modelle gerade erst an. In komplexeren, kreativen Aufgaben hatte eine künstliche Intelligenz bisher wenig Chancen gegen einen Menschen. Aber nun haben Wissenschaftler es sogar geschafft, eine künstliche Intelligenz zu entwickeln, welche in Coding-Wettbewerben, die unter anderem von Firmen zum Rekrutieren neuer Programmierer genutzt werden, in



Abb. 27: Manipulierte Verkehrsschilder als Beispiele für eine Adversarial Attack. ©[3]

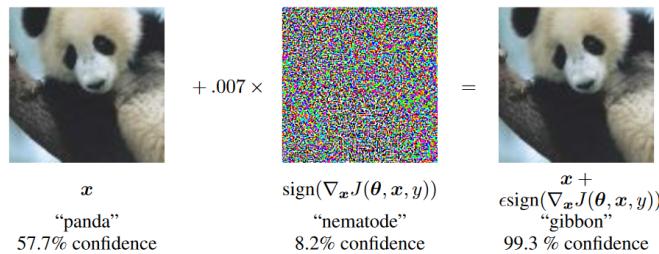


Abb. 28: Spezifisch berechnetes digitales Rauschen wird mit dem Bild überlagert, sodass eine völlig falsche Klassifizierung entsteht. Das letzte Bild ist eine Kombination der beiden vorherigen und wird mit einer Sicherheit von 99% als Gibbon erkannt. Die Bilder sind jedoch für das menschliche Auge nicht auseinanderzuhalten [6]. Dies zeigt, wie unterschiedlich Mensch und Maschine wirklich arbeiten. ©[6]

den oberen 54% abschneidet [10]. Dies sieht auf den ersten Blick eventuell nach keinem großen Erfolg aus, allerdings war es vor ein paar Jahren noch undenkbar, diese Aufgabe überhaupt von einer Maschine lösen zu lassen. Eine solche Aufgabe erfordert hohe Kompetenz im Problem-Lösen und eine eigene Kreativität, von deren Umsetzung bis vor kurzem nur geträumt werden konnte. Die Forschung an künstlicher Intelligenz entwickelt sich rasant voran, und es ist kaum möglich abzuschätzen, wozu Machine-Learning in ein paar Jahren in der Lage sein wird.

Literatur

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee, 2017.
- [3] Kevin Eykholt, Ivan Evtimov, Earlene Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1625–1634, 2018.
- [4] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc, 2019.
- [5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [6] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
- [7] Keras API reference. <https://keras.io/api/>. Aufgerufen: 2022-03-01.
- [8] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [9] Alex Krizhevsky. The CIFAR-10 dataset. <https://www.cs.toronto.edu/~kriz/cifar.html>. Aufgerufen: 2022-03-01.
- [10] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.
- [11] Navin Kumar Manaswi. Understanding and working with Keras. In *Deep Learning with Applications Using Python*, pages 31–43. Springer, 2018.
- [12] Installing packages. <https://packaging.python.org/en/latest/tutorials/installing-packages/>. Aufgerufen: 2022-02-26.

- [13] Neil Savage. How AI is improving cancer diagnostics. *Nature*, 579(7800):S14–S14, 2020.
- [14] Sun-Chong Wang. Artificial neural network. In *Interdisciplinary computing in java programming*, pages 81–100. Springer, 2003.
- [15] Wikipedia contributors. Kernel (image processing) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Kernel_\(image_processing\)&oldid=1071018204](https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&oldid=1071018204), 2022. [Online; accessed 24-February-2022].

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe.

Göttingen, den 22.3.2022

A handwritten signature in black ink, appearing to read "David Söding". The signature is fluid and cursive, with "David" on top and "Söding" below it.

David Söding

Erklärung

Hiermit erkläre ich, dass ich damit einverstanden bin, wenn die von mir verfasste Facharbeit der schulinternen Öffentlichkeit zugänglich gemacht wird.

Göttingen, den 22.3.2022

A handwritten signature in black ink, appearing to read "David Söding". The signature is fluid and cursive, with "David" on top and "Söding" below it.

David Söding