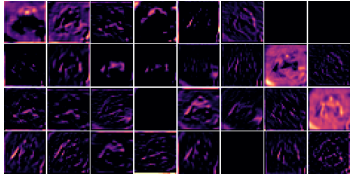
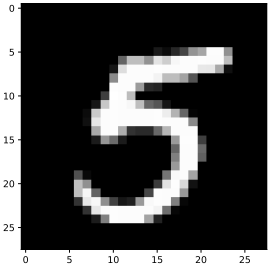
Convolutional Neural Networks

Prof. Dr. Jörg Frochte

Maschinelles Lernen

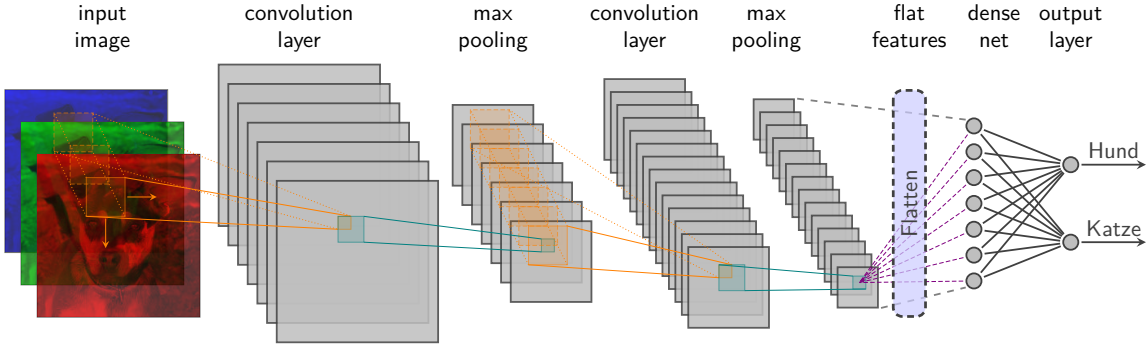


Schwierigkeiten der klassischen Netze



- Klassische MLP-Ansätze haben bei der Bilderkennung ihre Schwierigkeiten.
- Die Erkennung braucht eine gleichbleibende Ausrichtung der Bilder im Eingabevektor.
- Ist dies nicht der Fall, bricht die Genauigkeit deutlich ein.
- Der Grund ist, dass unser Netz keinen räumlichen Bezug hat. Jeder Pixel steht für sich allein und ist in einem Vektor aneinandergereiht.
- Wenn also in dem Bild eine kleine Translation um wenige Pixel stattfindet, bedeutet dies eine deutliche Auswirkung. Man kann schnell merken, dass die MLPs vergleichsweise empfindlich sind gegenüber Translationen, Rotationen und Skalierungen.
- Ein Ansatz damit umzugehen sind **2D Convolutional Neural Networks**.

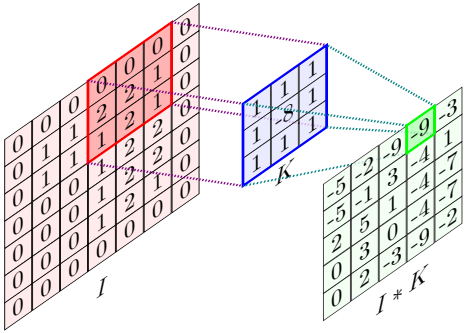
Übersicht: Convolutional Neural Networks (CNNs)



- Die Verarbeitung von Bilddaten ist die Hauptanwendung von CNNs.
- Die Struktur ist genau wie in 1D, nur dass das Ergebnis einer Faltung 2D ist.
- Dementsprechend haben die Daten eine Dimension mehr:
  Samples $\times$ Bildhöhe $\times$ Bildbreite $\times$ Kanäle

(Zero-)Padding

- Wir kennen schon das Problem an den Rändern von den Zeitreihen.
- Ohne Änderung wird aus einer 5x5 Matrix z.B. eine 3x3 Matrix nach der Faltung.
- Eine Technik um damit umzugehen ist das sogenannte **Zero Padding**

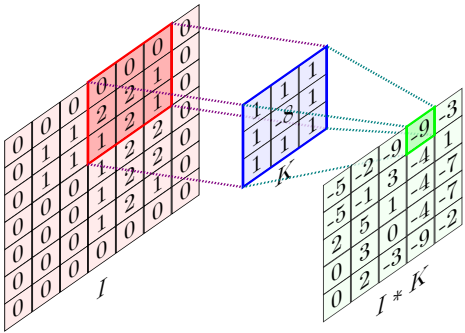


Faltung auf einem 2D-Array

- Das ist besonders bei kleinen Inputs interessant.
- Schneidet ein Kernel/Filter prozentual wenig von einem Bild an verzichtet man eher auf ein Padding.
- Natürlich muss der Filter sich auch nicht immer genau einen Pixel weit über das Bild bewegen.
- In Keras kann man dem Faltungslayer, den wir später noch kennenlernen werden, die Option **strides** übergeben.

Faltung mit einem bekannten Filter

- Folgender Beispielfilter ist bekannt für seine Eigenschaft Kanten zu betonen.
- Bei einem CNN werden die Filterkernel natürlich gelernt und geben immer nur ein Grauwertbild aus. Dabei hätte ein Filterkernel für ein RGB-Bild z. B. $(3 \times 3) \times 3$ Gewichte.

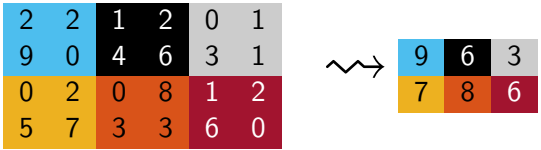


Faltung auf einem 2D-Array

Effekt auf ein Farbbild

Pooling

- Bei der Faltung und beim Pooling ergeben sich Probleme am Rand.
- Wenn das gefaltete Bild gleichgroß bleiben soll kann man außerhalb des Bildes Nullen verwenden (Zero Padding) mit dem Parameter `padding='same'`.
- Dies gilt beim Pooling `MaxPooling2D`, sowie für die Faltung `Conv2D`.
- In 2D fasst Max-Pooling natürlich auch zweidimensionale Bereiche zusammen.



(2,2) Max-Pooling

- Hierfür unterteilt man eine Feature Map in $y \times x$ große Rechtecke und übernimmt hierbei nur den größten Wert aus jedem Rechteck (häufig gilt $y = x$).
- x, y sind typischerweise klein und variieren zwischen 2, 3 und maximal 4.

Effekte von Faltung und Pooling am Beispiel CIFAR-10

- CIFAR-10 ist eine Sammlung von 32×32-Pixel RGB-Bildern.
- Sie enthält 60.000 Bilder, die zu zehn verschiedenen Klassen gehören.
- Von jeder Klasse sind 5000 im Trainingsset und 1000 im Testset.
- Eine Abfolge von Faltungen und Pooling-Schritten könnte z. B. so aussehen:
  (Standardisierung ist bei Bildern *unüblich*, der Mittelwert kann aber abgezogen werden)

Andere Datenbestände die wir neben CIFAR-10 und CIFAR-100 auch nutzen sind:

1. MNIST (Ziffern)
2. EMNIST (Buchstaben)
3. Fashion-MNIST (Symbole)
4. Cats and Dogs Dataset
5. ...

CIFAR-10 in Python

- CIFAR-.10 kann direkt inklusive Aufteilung über Keras bezogen werden.

```
1  import numpy as np
2  from tensorflow.keras.datasets import cifar10
3  from tensorflow.keras.utils import to_categorical
4  from tensorflow.keras.models import Sequential
5  from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten, Dense
6  from tensorflow.keras.regularizers import l2
7
8  (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()
9  noOfClasses = 10
10 YTrain = to_categorical(yTrain, noOfClasses)
11 YTest  = to_categorical(yTest, noOfClasses)
12 XTrain = xTrain/255.0
13 XTest  = xTest/255.0
14 l2Reg = 0.001
```

CIFAR-10 in Python

```
15
16 CNN = Sequential()
17 CNN.add(Conv2D(32,(3,3),padding='same',activation='relu',kernel_regularizer=l2(l2Reg),
18                 input_shape=(32,32,3)))
19 CNN.add(MaxPool2D(pool_size=(2, 2),padding='same'))
20 CNN.add(Conv2D(32,(3,3),padding='same',activation='relu',kernel_regularizer=l2(l2Reg)))
21 CNN.add(MaxPool2D(pool_size=(2, 2),padding='same'))
22 CNN.add(Conv2D(64,(3,3),padding='same',activation='relu',kernel_regularizer=l2(l2Reg)))
23 CNN.add(MaxPool2D(pool_size=(2, 2),padding='same'))
24 CNN.add(Conv2D(64,(3,3),padding='same',activation='relu',kernel_regularizer=l2(l2Reg)))
25 CNN.add(MaxPool2D(pool_size=(2, 2),padding='same'))
26 CNN.add(Flatten())
27 CNN.add(Dense(512,activation='relu',kernel_regularizer=l2(l2Reg)))
28 CNN.add(Dense(256,activation='relu',kernel_regularizer=l2(l2Reg)))
29 CNN.add(Dense(10,activation='softmax'))
30 CNN.summary()
```

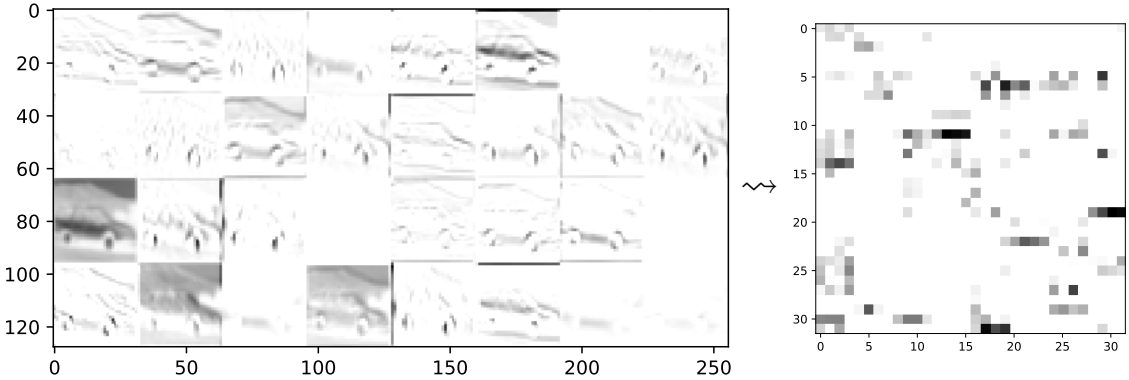
CIFAR-10 in Python

```
31 CNN.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
32 CNN.fit(XTrain,YTrain,epochs=20,batch_size=64)
33 scores = CNN.evaluate(XTest,YTest,batch_size=64)
34 print("Accuracy: %.2f%%" % (scores[1]*100))
35 yPred = CNN.predict(XTest)
36 choise = np.argmax(yPred, axis=1)
```

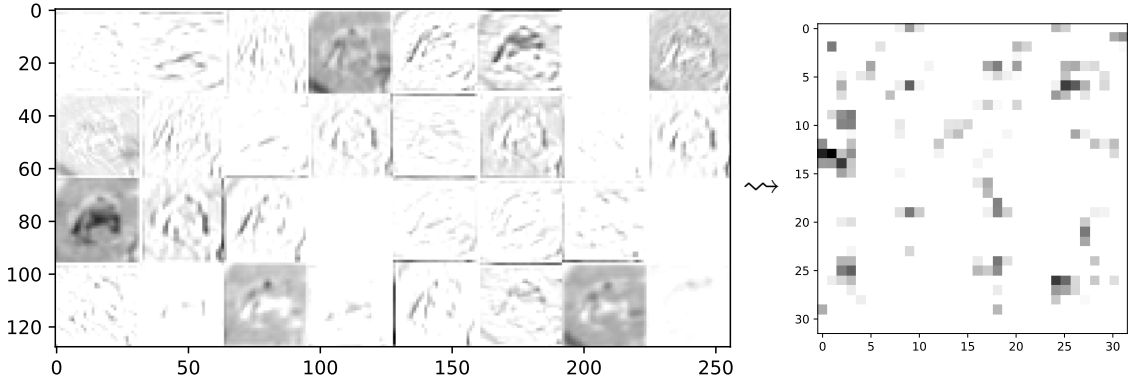
```
>> yPred[0,:]
        array([6.0782034e-04, 2.8072982e-04, 4.9193479e-02, 6.3261729e-01,
        6.7229304e-03, 2.2683969e-01, 7.7098981e-02, 2.6763529e-03,
        3.7491180e-03, 2.1360841e-04], dtype=float32)
```

- Die Ausgabe in einer Konfusionsmatrix spiegelt nicht wieder wie knapp ggf. die jeweilige Entscheidung war.
- Auch hier gilt es zusätzlich Unsicherheiten zu quantifizieren.

Ausgabe nach der ersten & letzten Faltung (Auto)



Ausgabe nach der ersten & letzten Faltung (Frosch)



Konfusionsmatrix (absolut)

	Flugzeuge	Autos	Vögel	Katzen	Rehe	Hunde	Frösche	Pferde	Schiffe	Lastwagen
Flugzeuge	840	14	42	23	10	6	8	6	21	30
Autos	29	837	5	15	1	6	5	0	28	74
Vögel	75	3	685	85	41	52	35	12	4	8
Katzen	23	9	79	648	34	127	45	16	10	9
Rehe	32	3	126	91	601	40	61	33	8	5
Hunde	16	1	59	194	21	660	15	23	7	4
Frösche	8	4	63	70	15	11	814	3	9	3
Pferde	20	2	47	79	40	77	7	716	4	8
Schiffe	102	25	14	21	1	5	2	2	798	30
Lastwagen	34	65	11	19	1	7	5	7	17	834

- Die Gesamtgenauigkeit liegt hier bei ca. 73% (geht besser).
- Der Fehler ist nicht gleichmäßig bzgl. der Klassen verteilt: Rehe werden am schlechtesten erkannt und Flugzeuge am besten.
- Die Konfusionsmatrix ist nicht symmetrisch.
- Hunde werden häufiger als Katzen klassifiziert als umgekehrt.

Vorteile eines CNN

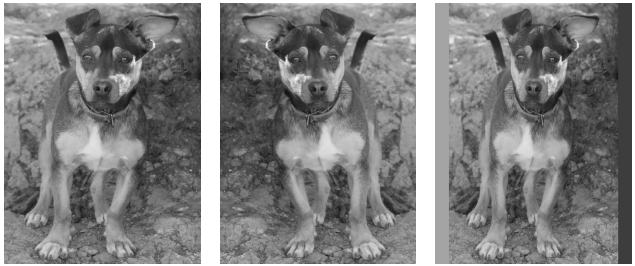
Die drei wichtigsten Eigenschaften für CNN sind:

- Dünnbesetzte Wechselwirkungen (**sparse interactions**):
  Die dünnbesetzte Wechselwirkungen ergeben sich aus den Kerneln mit ihrem beschränkten Wirkungsbereich.
- Gemeinsame Parameter (**parameter sharing**):
  Dieselben Parameter, also Faltungskernel werden überall im Bild angewendet.
- Äquivariante Repräsentationen (**equivariant representations**):
  Eine verschobenes Objekt in einem Bild (z. B. die Ziffer) wird dasselbe gefaltete Bild ergeben, nur eben verschoben (translationsinvariant).

Ein CNN ist aber nicht invariant gegenüber Drehungen, Spiegelungen, Größenskalierungen, etc. Diese müssen dem anders beigebracht werden.

Data Augmentation

- Ein gespiegelter Hund ist immer noch ein Hund, und wenn man ihn leicht zusammendrückt und schlanker macht, bleibt es auch ein Hund.
- Durch zufällige leichte Drehungen, Spiegelungen und Scherungen können so die Datenbestände aufgestockt werden.
- Das ist wichtig, um das Netz robuster zu machen und wirkt wie eine Regularisierung.
- Die Klasse `tensorflow.keras.preprocessing.image.ImageDataGenerator` kann solche Daten on-the-fly generieren.



Nutzung fertiger Netze bzw. Netzteile

- Es gibt die Möglichkeit den Rumpf eines CNN – also die Filter – für einen neuen Zweck wiederzuverwerten.
- Der Fully-connected Layer ist oft zu spezialisiert für eine Wiederverwendung.
- Keras bietet zu diesem Zweck einige bekannte Netze fertig trainiert an.
- Der folgende Code demonstriert, wie man z. B. `VGG16` verwenden kann, um ein eigenes Netz zu bauen.

```
from tensorflow.keras.applications import VGG16
convBase = VGG16(include_top=False, input_shape=(48, 48, 3))
```

- Der Parameter `include_top=False` bedeutet, dass auf den Fully-connected Layer verzichtet werden soll.
- Die möglichen Input-Dimensionen hängen vom Netz ab.

Nutzung fertiger Netze bzw. Netzteile

Nun nehmen wir einfach ein leeres Sequential-Model und fügen diesen fertigen Rumpf hinzu.

```
cnn = Sequential()
cnn.add(convBase)
```

Es gibt die Möglichkeit, in Keras einzelne Einheiten des Netzes vom Training auszusparen und damit die Gewichte des importierten Rumpfs fest zu lassen. In der `cnn.summary()` würden die Gewichte als *non-trainable* angezeigt.

```
convBase.trainable = False
```

Anschließend fügen wir den dichten Teil an und und trainieren diesen Teil:

```
cnn.add(layers.Flatten())
cnn.add(layers.Dense(512, activation='relu',  kernel_initializer='he_uniform'))
cnn.add(layers.Dense(10, activation='softmax'))
cnn.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
cnn.fit(XTrain, YTrain, epochs=10, validation_data=(XTest, YTest))
```

Nun könnte man das gesamte Netz mit Rumpf vorsichtig (kleine Lernrate) nachtrainieren.