HOCHSCHULE HEILBRONN

Hochschule für Technik, Wirtschaft und Informatik

Studiengang Wirtschaftsinformatik (WIN)

Fallstudie – Entwicklungswerkzeuge

Machine Learning – TensorFlow

vorgelegt bei

Prof. Dr. Mahsa Fischer

von

Yacin Boualili (200808)

&

Valentin Weber (200801)

im

Wintersemester 2019/20

Inhaltsverzeichnis

Abbildungsverzeichnis

Abstract

In diesem Papier soll das Framework „TensorFlow“ des Google Brain Teams anhand der praktischen Implementation eines neuronalen Netzes erklärt werden. Hierzu werden zunächst die Begrifflichkeiten „künstliche Intelligenz“, „maschinelles Lernen“ und „neuronale Netze“ voneinander abgegrenzt.

Anschließend beginnt bereits die Implementation des Netzes. Hierbei wird zuerst auf die Vorbereitung der Datengrundlage eingegangen und Konzepte wie Trainings- und Validierungsdaten näher beleuchtet, bevor es an den tatsächlichen Aufbau des neuronalen Netzwerks geht.

Hierzu wird die Funktionalität jeder verwendeten Layerart („Convolutional“, „Pooling“, „Drop“, „Flatten“ und „Dense“) einzeln erläutert, sowie anhand des Beispiels aufgezeigt, wie ein solcher Layer mithilfe des TensorFlow Frameworks implementiert wird.

Nachdem das Model definiert wurde, wird näher auf den Trainingsprozess, insbesondere auf die Werte „Loss“ und „Accuracy“, sowie die Phänomene „Over-“ bzw. „Underfitting“ eingegangen.

Abschließend wird aufgezeigt, wie das trainierte Model genutzt und die Output Daten interpretiert werden.

Einleitung

Der Begriff künstliche Intelligenz ist seit einigen Jahren auch im Mainstream, also bei weniger technikaffinen, Zielgruppen immer bekannter. Häufig werden dabei allerdings, mangels der technischen Kenntnisse, dazugehörige Begriffe wie maschinelles Lernen und neuronale Netze synonym verwendet. Dieses Papier soll die Begriffe voneinander abgrenzen und anhand des Frameworks TensorFlow erklären, wie diese Begriffe miteinander zusammenhängen. Zu diesem Zweck müssen zunächst einige grundlegende Fragen geklärt werden, damit die Relevanz von TensorFlow aufgezeigt werden kann.

Was ist maschinelles Lernen?

Unter maschinellem Lernen wird die Gewinnung von Wissen mithilfe von, Erfahrungen verstanden, die durch ein künstliches System gesammelt wurden. Hierzu bauen Algorithmen auf Grundlage von Beispielen ein statistisches Modell auf, in dem Muster in den Beispieldaten gefunden werden, um das Gelernte später auf komplett neue Datensätze anwenden zu können. Maschinelles Lernen ist also ein Ansatz, mit dem eine künstliche Intelligenz erschaffen werden kann, ist aber nicht gleichbedeutend mit ihr.

Die zugrundeliegenden Algorithmen, durch welche das Lernen praktisch umgesetzt wird, lassen sich in zwei übergeordnete Kategorien einteilen. Das überwachte und das unüberwachte Lernen. Letzteres dient der Klassifizierungen von Datenclustern und der Ermittlung von Zusammenhängen in diesen, zur Erstellung eines statistischen Modells, welches Vorhersagen auf Grundlage der erkannten Zusammenhänge ermöglicht. Das unüberwachte Lernen hingegen beschreibt einen Ansatz, bei dem der Algorithmus die Klassifikation der Daten nicht selbst übernehmen wird, weil diese ihm durch eine „Supervisor“ als Funktionsinput mitgeliefert werden. So kann der Algorithmus nach wiederholter Analyse der Beispieldaten eigene Assoziationen herstellen, um so auch bei unbekannten Daten die richtigen Entscheidungen zu treffen.

Ein Ansatz zur Implementation solcher Algorithmen des maschinellen Lernens stellen neuronale Netze dar. TensorFlow ist ein Framework, welches den Aufbau solcher Netze für den Entwickler vereinfachen soll. Doch bevor geklärt wird was TensorFlow ist und wie man es einsetzt, muss zunächst geklärt werden, was unter dem Begriff neuronales Netz überhaupt verstanden wird.

Was ist ein neuronales Netz?

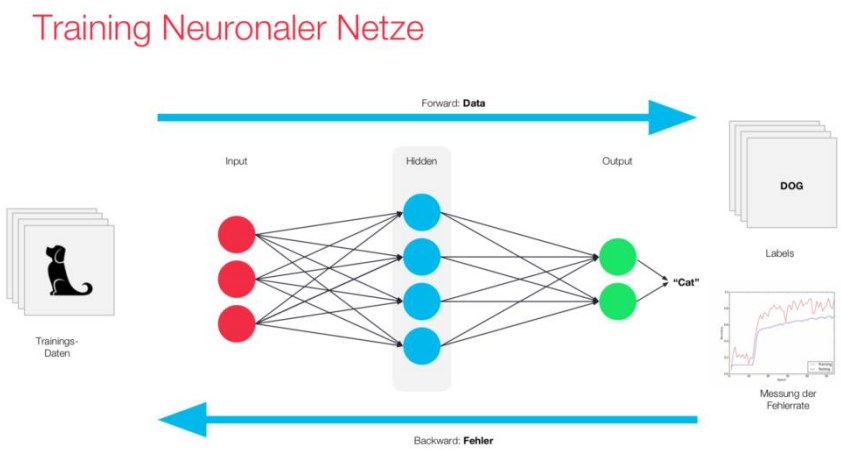


Abbildung 1: Aufbau eines sehr einfachen neuronalen Netzes

Neuronale Netze sind im Gespräch, seit programmierbare Computer im Bereich der angewandten Mathematik eingesetzt wurden, also bereits seit den frühen 1940er Jahren. Der Begriff beschreibt ein System aus Algorithmen, welches zur Implementation von maschinellem Lernen verwendet werden kann. Hierzu durchlaufen die Daten, mit denen das Netz trainiert werden soll, mehrere Schichten (bzw. Layer) in denen sich die Neuronen befinden. Diese Neuronen repräsentieren wiederum Algorithmen, welche die Eingabedaten tatsächlich verarbeiten und über Pfade miteinander verbunden ist. Diese einzelnen Pfade sind gewichtet und der einzuschlagende Weg hängt von den Berechnungsergebnissen der Neuronen ab, sodass am Ende aufgrund des gegangenen Pfades ein eine endgültige Entscheidung über die Eingabedaten getroffen werden kann.

Neuronale Netze können in zwei Kategorien unterteilt werden, die je nach Anwendungszweck und verfügbaren Beispieldaten eigene Vor- und Nachteile bieten. Zum einen gibt es die Konvolutionsnetze, welche rein vorwärtsgerichtet arbeiten und beliebig viele versteckte Layer, also nicht Ein- bzw. Ausgabelayer, beinhalten können, in denen der Entscheidungsfluss nur in eine Richtung läuft. Zum anderen gibt es rekurrente neuronale Netze, welche auch Rückkopplungen, also einen rückwärtsgewandten Datenfluss, beinhalten können, wodurch dem System eine Art „Gedächtnis“ mitgegeben wird.

Was ist TensorFlow?

TensorFlow ist ein Framework zur Bildung neuronaler Netzwerke, um maschinelles Lernen zu implementieren. Entwickelt wurde es, zunächst zur rein internen Nutzung, durch das Google Brain Team und wurde im November 2015 unter der Apache Licence v2 als OpenSource Framework veröffentlicht. TensorFlow selbst ist in C++ implementiert, bietet aber schnittstellen zu verschiedensten Programmiersprachen, aus denen

es bedient werden kann. Am besten unterstützt wird hierbei die Sprache Python, es ist aber auch durch C, C++, Java, JavaScript oder Go ansprechbar und ist deshalb in vielen Anwendungsbereichen nutzbar. Durch die Veröffentlichung als OpenSource Software konnten Drittanbieter weitere Bibliotheken zur Anbindung von TensorFlow in weiteren Programmiersprachen entwickeln und somit die Vielseitigkeit des Frameworks weiter steigern.

Zusätzlich zu der vielseitigen Anbindung an verschiedene Programmiersprachen bietet TensorFlow auch Anbindungen an andere beliebte Software aus dem Bereich maschinelles Lernen wie zum Beispiel die Bibliothek Keras bzw. oder der Programmiertechnik CUDA, ein von NVIDIA bereitgestelltes Framework zur Ausführung von Programmteilen auf einer Grafikkarte (GPU), wodurch insbesondere parallele Abläufe effizienter abgearbeitet werden können.

Diese vielfältige Unterstützung hat zur Folge, dass TensorFlow eines der beliebtesten Frameworks zur Umsetzung von neuronalen Netzen ist und in verschiedensten Branchen erfolgreich kommerziell genutzt wird (siehe Abbildung 2).

Abbildung 2: Kommerzielle Verwendung von TensorFlow

Mithilfe von TensorFlow können sowohl Konvolutions- als auch rekurrente Netze implementiert werden. Dieses Papier wird sich allerdings auf Konvolutionsnetze beschränken und anhand eines Beispielsnetzes, welches Hunde- und Katzenbilder erkennen soll, verschiedene Layerarten sowie deren konkrete Implementation mithilfe eines Jupyter Notebooks erläutern.

Voraussetzungen

Um TensorFlow mit Python verwenden zu können, muss die Sprache in der 64 Bit Version 3.7 sowie alle nötigen Abhängigkeiten installiert werden. Je nach Anwendungsfall und Datengrundlage können verschiedenste Bibliotheken zur Datenaufbereitung verwendet werden. Die Installation kann entweder über die Kommandozeile oder über die Entwicklungsumgebung PyCharm erfolgen. In diesem Papier wird lediglich die Einrichtung über die Kommandozeile beschrieben, denn die Einrichtung über PyCharm funktioniert analog zu der über die Kommandozeile.

Benötigte Bibliotheken

Für unser Fallbeispiel, in dem ein neuronales Netz zur Erkennung von Hunde- bzw. Katzenbildern implementiert werden soll, werden die folgenden Bibliotheken benötigt:

- tensorflow
  - tensorflow wird in der Version 1.14 benötigt, hierauf ist bei der Installation explizit zu achten, weil bereits die Version 2.0 erschienen ist
- jupyter
  - wird benötigt, um das Jupyter Notebook ausführen und darstellen zu können. Das Notebook bietet eine anschauliche Oberfläche, anhand derer einzelne Programmschritte deutlicher herausgestellt werden können, wodurch der Prozess leichter nachvollziehbar wird.
- numpy
  - numpy wird verwendet, um die gelesenen Bilddaten in ein Format zu bringen, welches von Tensorflow verstanden werden kann
- matplotlib
  - matlpotlib dient der Darstellung der eingelesenen und mithilfe von numpy formatierten Bildern, um einfacher nachvollziehen zu können, wie die Formatierung der Bilddaten geschiet
- pillow
  - pillow, oder PIL (Python Image Library) dient dem einlesen und skalieren der Bilddaten, welche das neuronale Netz später analysieren soll.

Einrichtung über die Kommandozeile

Zunächst muss ein Ordner erstellt werden. Der Name des Ordners ist irrelevant, für die weiteren Einrichtungsschritte wird aber davon ausgegangen, dass der Ordner „FSEW_Tensorflow“ genannt wird. Die Struktur des Ordners sollte wie folgt aussehen:

- FSEW_Tensorflow
  - Data
  - Logs
  - Models

Nun sollten die für das Training benötigten Bilder heruntergeladen werden. Hierzu wird das Trainingsset *Kaggle Cats and Dogs* von Microsoft verwendet.[1] In dem heruntergeladenen ZIP-Archiv befinden sich zwei Ordner mit den Namen „Cats“ und „Dogs“. Diese Ordner müssen in den soeben erstellten Unterordner Data kopiert werden.

Kommentiert [VW1]:

Nun sollte eine virtuelle Umgebung erstellt werden, in der die einzelnen Abhängigkeiten installiert werden, wie das funktioniert wird auf der folgenden Abbildung erklärt:

[1] https://www.microsoft.com/en-us/download/confirmation.aspx?id=54765

1. Virtuelle umgebung erstellen, aktivieren und Bibliotheken installieren

1. Konsole öffnen und in den Ordner `FSEW_Tensorflow` navigieren
    - Dateipfad im Explorer/Finder kopieren und `cd <kopierter Pfad>` in Konsole eingeben
2. Neue Virtuelle Umgebung erstellen
    - **Windows:** `py -m venv .\venv`
    - **Mac:** `python3 -m venv .\venv` (evtl. python statt python3)
3. Virtuelle Umgebung aktivieren
    - **Windows**: `venv\Scripts\activate.bat`
    - **Mac:** `venv\bin\activate`
4. Bibliotheken installieren
    - pip & setuptools updaten:
        - **Windows**:: `py -m pip install --upgrade pip setuptools`
        - **Mac:** `python -m pip install --upgrade pip setuptools`
    - Die Bibliotheken können mit dem Befehl: `pip install {BIBLIOTHEK}` installiert werden

Abbildung 3: Einrichtung der virtuellen Umgebung

Nach erfolgreicher Installation der Bibliotheken müssen zwei Konsolenfenster geöffnet werden, um die lokalen Server für das Jupyter Notebook sowie den lokalen Tensor-Board Server zu starten:

2. Jupyter Notebook starten

1. Im Konsolenfenster `jupyter notebook` eingeben
    - Es sollte sich ein Browserfenster öffnen

3. TensorBoard Server starten

1. Neues Konsolenfenster öffnen und Virtuelle Umgebung starten
    - siehe Schritte 1.1 und 1.3
    - **WICHTIG:** Beide Schritte einzeln ausführen
2. `tensorboard --logdir Logs` eingeben
    - In einem neuen Browsertab `localhost:6006` öffnen
    - Hier dürften noch keine Daten zu sehen sein, das kommt später

Abbildung 4: Starten von Jupyter & Tensorboard

Nach dem Start des Jupyter Servers sollte sich automatisch ein Browserfenster öffnen, in dem der Ordner FSEW_Tensorflow geöffnet ist. In diesem Ordner muss ein neues Jupyter Notebook erstellt werden. Dann ist die Einrichtung abgeschlossen.

Workshop: Katzen und Hundebilder

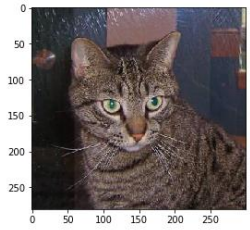
Als allererstes müssen, wie in jedem anderen Python Programm auch, die benötigten Module und Pakete der installierten Bibliotheken importiert werden, um diese im Code nutzen zu können. Für unser Notebook brachen wir die folgenden Bibliotheken:

```
import os
import time
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from random import shuffle
import tensorflow as tf
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Dropout,Conv2D,MaxPooling2D,Flatten
from tensorflow.python.keras.callbacks import TensorBoard
from tensorflow.python.keras.callbacks import ModelCheckpoint
```

Abbildung 5: Import der benötigten Bibliotheken

Nachdem die Bibliotheken erfolgreich importiert wurden, kann damit angefangen werden, die Datensätze zu überprüfen. Das ist notwendig, um sicherzugehen, dass sich unsere Katzen und Hundebilder in den erwarteten Verzeichnissen befinden. Ist dies nicht der Fall, muss hier angepasst werden, ansonsten würde das neuronale Netz später falsch trainiert.

```
In [2]: img = Image.open('Data/Cat/1.jpg')
        img.load()
        plt.imshow(img)
        plt.show()
```




```
In [3]: img = Image.open('Data/Dog/1.jpg')
        img.load()
        plt.imshow(img)
        plt.show()
```

Abbildung 6: Überprüfung der korrekten Ordner- & Datenstruktur

Haben wir sichergestellt, dass sich die Bilder da befinden, wo wir sie erwarten, können wir mit der Aufbereitung der Daten für das neuronale Netz beginnen.

Bilddaten skalieren

Wie an den beiden Beispielbildern gesehen, können die Bilder eingelesen und dargestellt werden, allerdings fällt auf, dass die Bilder verschiedene Auflösungen haben. Ein neuronales Netz muss aber mit gleich strukturierten Daten trainiert werden. Im Falle von Bildern bedeutet das, dass alle Bilder die gleiche Auflösung haben müssen. Um den benötigten Speicherplatz zu reduzieren, kann es außerdem sinnvoll sein, das Bild

von einer Farbaufnahme in ein Graustufenbild umzuwandeln, dadurch wird die Anzahl der Farbkanäle des Bildes von drei auf einen reduziert.

```
img = Image.open('Data/Cat/1.jpg').convert("I")
img = img.resize((50,50) , Image.LANCZOS)
img.load()
plt.imshow(img)
plt.show()
```

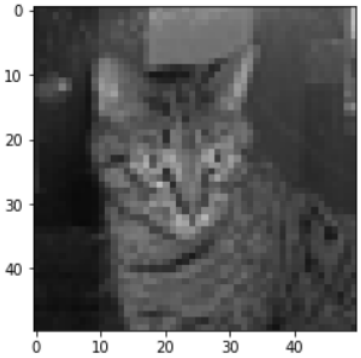


Abbildung 7: Skaliertes Katzenbild

Trotz des Informationsverlusts ist nach wie vor erkennbar, dass auf dem Bild eine Katze dargestellt wird. Nichtsdestotrotz hat sich der benötigte Speicherplatz signifikant reduziert, was die Effizienz des Trainings bei mehreren tausend Bildern steigert.

Im nächsten Schritt ist es notwendig, dass Bild in ein Format zu bringen, welches das neuronale Netz auch aufnehmen kann. Hierzu wird das Bild in ein Numpy Array umgewandelt, bei dem jeder Pixel des Bildes durch ein Element in dem Array repräsentiert wird. Dieses Element hat einen Zahlenwert zwischen 0 und 255, der den Farbwert dieses Pixels repräsentiert.

Wie in jedem anderen Bereich der Datenanalyse sollten diese Werte normalisiert, also in einen Wertebereich zwischen 1 und 0 gebracht, werden. Dies geschieht ganz einfach durch Division des Arrays durch 255. Anschließend muss die verlorengegangene Information der Anzahl von Farbkanälen wiederhergestellt werden. Hierzu wird in numpy die Funktion „reshape“ verwendet, welche das Array in die von uns gewünschte Struktur bringt.

```
data = np.asarray( img, dtype="int32" ).reshape((50,50,1))
data = data/255
print(data)
print(data.shape)
```

```
[[[0.16078431]
  [0.18431373]
  [0.16078431]
  ...
  [0.34901961]
  [0.4       ]
  [0.73333333]]
```

Da unser Netzwerk mit mithilfe eine überwachten Trainings lernen soll, ist es notwendig, den einzelnen Bildern ein Label mitzugeben. Die Anzahl der Labels ist dabei direkt von der Anzahl der verschiedenen Entscheidungsmöglichkeiten abhängig. In unserem Fall gibt es demnach die Labels „Cat“ und „Dog“. TensorFlow aber möchte die Labels in numerischer Form haben, wobei das erste Label immer die Zahl 0 trägt. In unserem Fall wird das Label „Cat“ durch eine 0, das Label „Dog“ durch eine 1 repräsentiert.

Diese Arbeitssschritte müssen nun für jedes einzelne Bild durchgeführt werden, dazu muss der folgende Codeblock sowohl für den Ordner *data/Cats* als auch *data/Dogs* durchgeführt werden. Die Funktion preprocessing() führt dabei die oben aufgelisteten Arbeitsschritte aus, während die Funktion labeling() anhand des übergebenen Labels eine 0 oder eine 1 zurückliefert. Die jeweiligen numpy Arrays für die einzelnen Bilder werden in einer Liste gespeichert, die später an das Model übergeben wird.

```
cat_data = []

Cats_dir = "Data/Cat"
for img in os.listdir(Cats_dir):
    try:
        data = preprocessing(img,Cats_dir)
        label = "Cat"
        target = labeling(label)
        cat_data.append([data,np.array(target)])
    except:
        continue
```

Abbildung 8: Umwandlung aller Katzenbilder

Zu allerletzt müssen wir sichergehen, dass unsere beiden Datensets die gleiche Anzahl an Elementen besitzen. Hierzu ermitteln wir welche Liste die geringere Anzahl an Elementen aufweist und bilden zwei gleich lange Listen auf Grundlage dieser Zahl.

```
min_len = min(len(cat_data),len(dog_data))
print(min_len)
```

```
cat_data = cat_data[:min_len]
dog_data = dog_data[:min_len]
print(len(cat_data))
print(len(dog_data))
```

Abbildung 9: Sicherstellung gleich vieler Elemente pro Datenset

Nun haben wir die benötigten Datensets eingelesen und für die Analyse durch das neuronale Netz vorbereitet. Nun müssen wir die Daten noch in Trainings- & Validierungsdaten aufteilen.

Trainings- und Validierungsdaten

Ein Konvolutionsnetz arbeitet immer mit zwei verschiedenen Datensets. Den Trainingsdaten einerseits, sowie den Testdaten andererseits. Während jedes Modeldurchlaufs werden alle Trainingsdaten verarbeitet, um die Gewichtungen der Pfade zwischen einzelnen Neuronen anzupassen, die Validierungsdaten hingegen dienen der Überprüfung der ermittelten Gewichtungen. Hierbei gilt als Faustregel, dass 70% der Daten zum Training, 30% zur Validierung verwendet werden. Hierzu bilden wir pro Datenset zwei neue Listen für Trainings- und Validierungsdaten. Anschließend werden die Daten für Katzen und Hunde zusammengefügt, vermischt und die Numpy Arrays aus den entsprechenden Listen gezogen.

```
train_max = int(0.7 * min_len)
print(train_max)
```

```
cat_train = cat_data[:train_max]
cat_val = cat_data[train_max:]
dog_train = dog_data[:train_max]
dog_val = dog_data[train_max:]
```

```
training_data = cat_train + dog_train
validation_data = cat_val + dog_val
```

```
shuffle(training_data)
shuffle(validation_data)
```

```
X = np.array([i[0] for i in training_data])
Y = np.array([i[1] for i in training_data])
val_x = np.array([i[0] for i in validation_data])
val_y = np.array([i[1] for i in validation_data])
```

Nun sind alle benötigten Bilddateien eingelesen, skaliert und in Trainings- bzw. Validierungssets eingeteilt. Jetzt geht es daran mithilfe von TensorFlow das neuronale Netz zu konfigurieren.

Aufbau und Konfiguration des neuronalen Netzes

Das neuronale Netz, in Zukunft auch als Model bezeichnet, wird in Python durch eine Klasse repräsentiert. Für dieses Fallbeispiel wird das Sequential Model der Keras Bibliothek verwendet, welche in TensorFlow mitgeliefert wird. Die Model Klasse wird zur Initiierung eine Liste an Layern übergeben. Diese Layer werden im Code auch durch Klassen repräsentiert, welche je nach Layerart unterschiedliche Parameter erwarten. In diesem Abschnitt sollen die einzelnen Layerarten, welche für unser Model benötigt werden, detailliert beschrieben, damit ihre Funktionalität und die Notwendigkeit besser

nachvollzogen werden kann. Hierzu müssen aber zunächst einige Hyperparameter festgelegt werden.

Hyperparameter festlegen

Die Hyperparameter legen die Rahmenbedingungen fest, unter denen das Model trainiert werden soll. Der Einfachheit halber werden in diesem Model für jeden Layer die gleichen Parameter verwendet, bei komplexeren Modellen allerdings kann es sinnvoll sein, die Parameter für jeden Layer einzeln zu definieren. In diesem Modell werden die folgenden Hyperarameter benötigt:

```
epoch = 30
num_filters= 10
filter_size = 5
pool_size = 5
DROP = 0.2
DECAY = 1e-5
LR = 0.001
```

Abbildung 10: Festlegung der Hyperparameter

Die genaue Bedeutung der einzelnen Parameter wird im Folgenden noch genauer erläutert werden.

Konvolutionslayer

Ein Konvolutionslayer dient der Analyse eines einzelnen Bildes. Hierzu bedient es sich sogenannter Filter, mit denen einzelne besonders aussagekräftige Eigenschaften im Bild gefunden werden sollen. In einem Konvolutionslayer beinhaltet jedes einzelne Neuron einen eigenen Filter, deren Anzahl und Größe frei wählbar sind. In dem Beispiel (siehe Abbildung 11) wird ermittelt, dass die Eigenschaften Diagonale (Filter 1 und 3), sowie deren Kreuzung (Filter 2), als Indiz dafür gewertet, dass auf dem Bild ein *X* dargestellt wird. Die Filter beinhalten dabei eine numerische Repräsentation des erwarteten Werts der einzelnen Pixel, im Beispiel repräsentiert die 1 einen Teil des *X*, während die -1 einen anderen Bildteil repräsentiert.

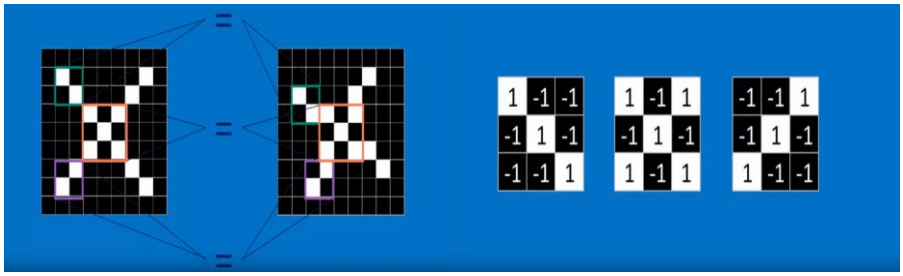


Abbildung 11: Beispiel für einen Filter im Konvolutionslayer

Um nun zu ermitteln, wie gut diese Filter auf das Bild passen, wird jedes einzelne Pixel des Bildes von jedem Filter analysiert. Hierbei wird überprüft, wie gut der analysierte Teil des Bildes mit dem Filter übereinstimmt. Hierzu wird die numerische Repräsentation des Pixels im zugrundeliegenden Bild mit den entsprechenden numerischen

Werten aus den Filtern multipliziert (Abbildung 12). Anschließend wird der Durchschnittswert all dieser Multiplikationen gebildet (Abbildung 13). Dieser Durchschnittswert wiederum repräsentiert, wie gut der Filter auf diesen Bildausschnitt passt.

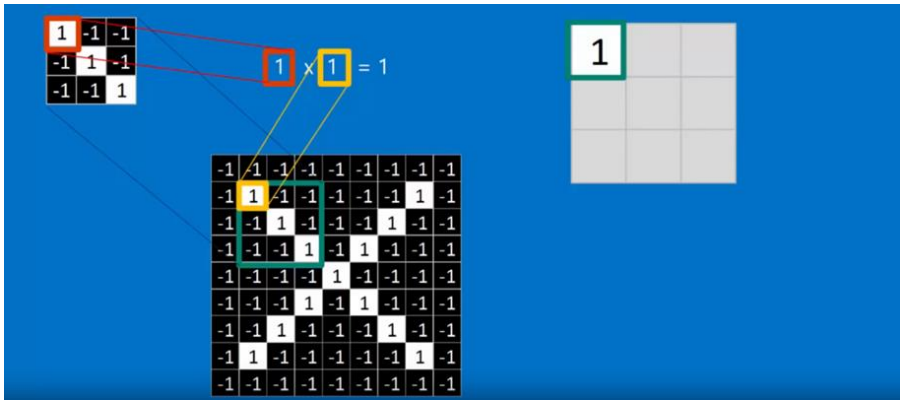


Abbildung 12: Abbildung des Filters auf einen Teil eines Analyseobjekts

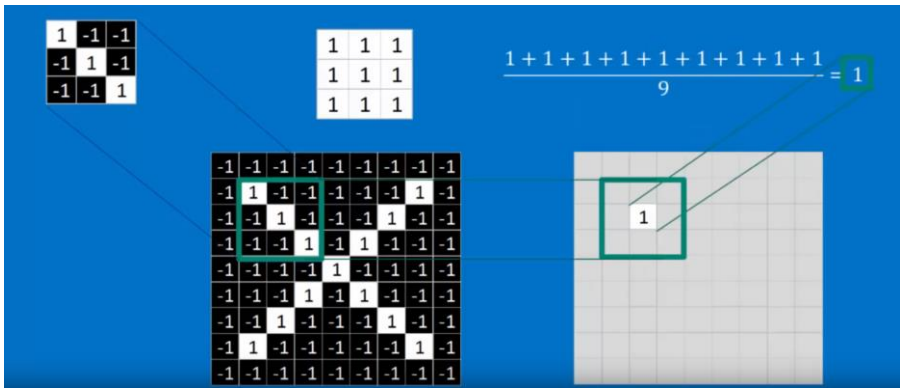


Abbildung 13: Auswertung der Abbildung des Filters auf einem Analyseobjekt

Dieser Prozess wird für jeden Filter durchgeführt und die Ergebnisse in eine Art Heatmap eingetragen. Der Output eines solchen Konvolutionslayers ist demnach eine Matrix, die für jeden einzelnen Pixel des Bildes angibt, wie gut welcher Filter auf diesen Bildausschnitt passt (Abbildung 14). Auf der Abbildung ist bereits gut zu erkennen, dass die einzelnen Filter gut auf das zu analysierende Bild anzuwenden ist. Dies entspricht vollkommen der Erwartung, da das analysierende Bild, wie für einen Menschen unschwer zu erkennen ist, ein *X* darstellt.

Abbildung 14: Ergebnis der Auswertung aller Filter eines Konvolutionslayers

Zur Weiterverarbeitung der ermittelten Daten wird in vielen Layern eine Aktivierungsfunktion benötigt, welche den letztendlichen Output des Layers so aufbereitet, dass dieser als Input für den nächsten Layer verwendet werden kann. In diesem Beispiel wird die Aktivierungsfunktion „Rectified Linear (relu)“ verwendet, welche dafür sorgt, dass alle negativen Werte aus der Heatmap, also diejenigen Felder, in denen der Filter nicht passt, auf 0 gesetzt werden (Abbildung 15).

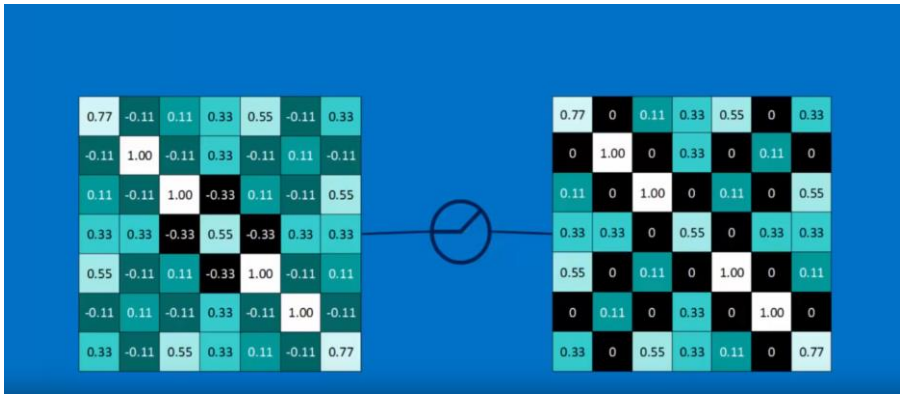


Abbildung 15: Anwendung der Aktivierungsfunktion "relu"

Bei der Verwendung von Aktivierungsfunktionen ist darauf zu achten, dass ein Input Layer, also der erste Layer des neuronalen Netzes, keine Aktivierungsfunktion benötigt, da die eingegebenen Daten bereits in einer Art und Weise skaliert wurden, dass der Layer mit ihnen arbeiten kann.

Entsprechend der definierten Hyperparameter für unser Modell befinden sich in jedem Konvolutionslayern zehn Filter, die jeweils eine Größe von 5 x 5 Pixeln besitzen. Handelt es sich nicht um den Input Layer, wird bei der Initiierung des Layerobjekts

zusätzlich der Parameter für die Aktivierungsfunktion mitgegeben. Zusätzlich muss die Input Shape angegeben werden, die dem Layer mitteilt, welche Form das zu analysierende Numpy Array besitzt. Zu allerletzt wird noch eine Padding Methode festgelegt, die definiert, wie der Output formatiert werden soll. Mit der hier angegebenen Methode „same“ ist die output_shape gleich der input_shape (Abbildung 16).

```
Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same"),

Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same",activation="relu"),
```

Abbildung 16: Input Layer (1) und Hidden Convolutional Layer (2) in TensorFlow

Nach der Definition der Konvolutionslayer müssen die Poolinglayer definiert werden, welche das Analyseobjekt weiter abstrahieren, um allgemeingültigere Aussagen treffen zu können.

Poolinglayer

In einem Poolinglayer wird das zu analysierende Bild weiter verkleinert, um das Model resistenter gegen Abweichungen innerhalb der einzelnen Bilder werden zu lassen. Dabei gehen die Pools grundsätzlich ähnlich vor, wie die Filter in den Konvolutionslayern.

Genau wie die Filter besitzt jeder Pool eine Größe, welche angibt, wie viele Pixel gepoolt werden sollen. Jeder Pool analysiert wiederum das gesamte Bild, dieses mal allerdings nicht jeden einzelnen Pixel, sondern lediglich jedes Bildsegment, dass der Größe des Pools entspricht.

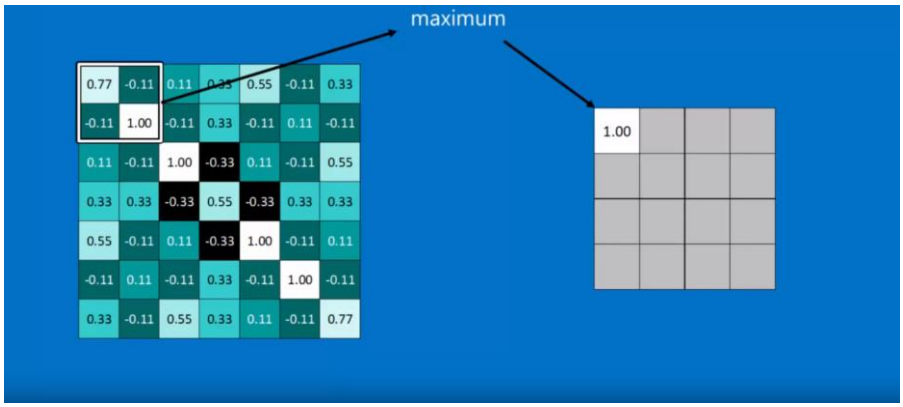


Abbildung 17: Anwendung eines Pools auf den Output eines Konvolutionslayers

Im Beispiel (Abbildung 17) wird ein Pool der Größe 2 angewendet. Der Pool umfasst demnach 4 Pixel und analysiert jedes 2x2 Quadrat der zugrundeliegenden Heatmap. Der Pool ermittelt hierbei den Maximalen Übereinstimmungswert des zu untersuchenden Bildes und baut daraus ein neues Bild auf, dies wird wiederum für jeden einzelnen Filter durchgeführt, sodass der Output eines Poolinglayers verkleinerte Versionen der Heatmaps sind, die ein Konvolutionslayer ermittelt hat (Abbildung 18).

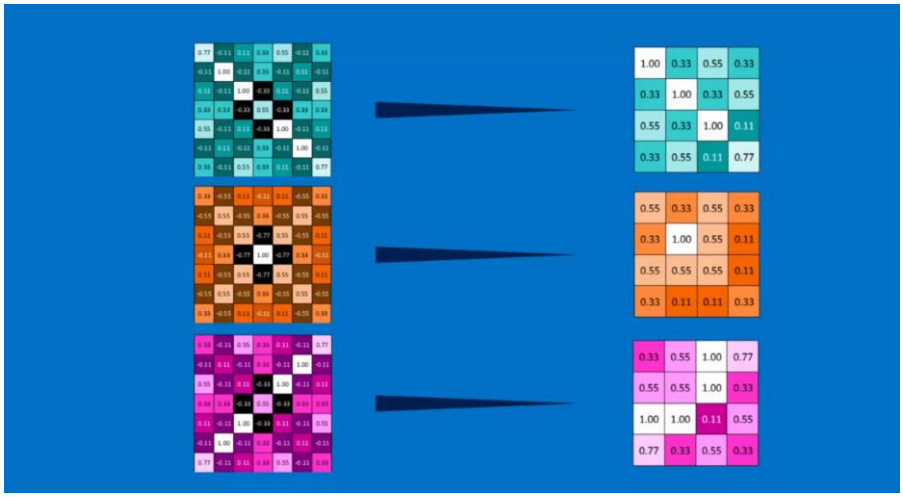


Abbildung 18: Beisielhafter Output eines Poolinglayers

Pooling Layer benötigen keine Aktivierungsfunktion. Sie berechnen nichts, sondern ermitteln lediglich das Maximum von zuvor errechneten Werten, welche bereits durch eine Aktivierungsfunktion normalisiert wurden. Konvolutionslayer und Poolinglayer können beliebig miteinander kombiniert werden, dabei ist die typische Vorgehensweise, dass zunächst ein Konvolutionslayer mit einer Aktivierungsfunktion und einem anschließenden Poolinglayer definiert werden (Abbildung 19).

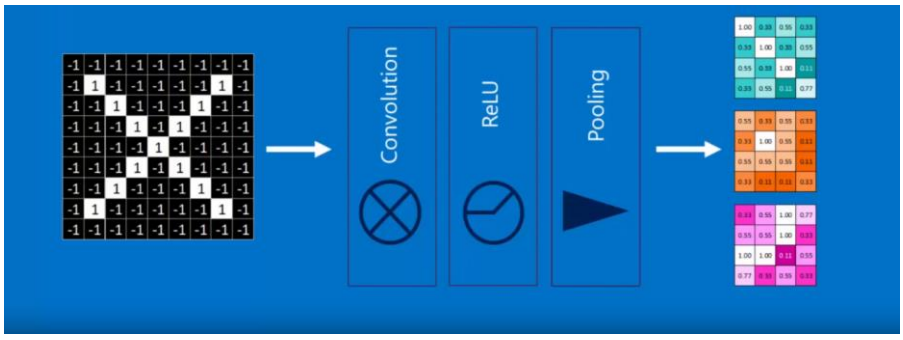


Abbildung 19: Typische Anordnung von Konvolutions- und Poolinglayern

Entsprechend der zuvor definierten Hyperparameter besitzen die Pools in unseren Layern eine Größe von 5 x 5 Pixeln. In Kombination mit der Filtergröße, die ebenfalls 5 x 5 Pixel beträgt, wird also der maximale Übereinstimmungswert in einem 25 Pixel umfassenden Quadrats ermittelt. Die Initiierung eines Poolinglayers im Python Code ist auf der folgenden Abbildung zu sehen.

```
MaxPooling2D(pool_size=pool_size, padding="same"),
```

Abbildung 20: Definition eines Poolinglayers mithilfe von TensorFlow

Wie bereits erwähnt, kann diese Abfolge von Konvolutions- und Poolinglayern beliebig oft wiederholt werden. Dabei gilt, dass je mehr Layer, Filter und Pools vorhanden sind, desto spezifischere Features können ermittelt werden. Je nach Anwendungsfall und Datengrundlage kann dies sowohl negative als auch positive Folgen haben, weswegen die beste Kombination von Layern und Filter- bzw. Poolgrößen in der Praxis meist durch Trial-and-Error ermittelt werden.

Drop Layer

Der Drop Layer dient dazu, einzelne Gewichtungen zu ignorieren, damit verschiedene Neuronen dieselben Konzepte lernen und so ein höherer Abstraktionsgrad des Models erreicht werden kann. Dies hilft insbesondere um gegen Overfitting vorzugehen, bei dem das Model immer und immer wieder dieselben Neuronen für die Entscheidungsfindung verwendet, sodass Muster nicht unbedingt erkannt werden können. So kann das Modell die zugrundeliegenden Trainingsdaten zwar schnell und sehr genau erkennen, bei stärkeren Abweichungen sind die Muster zur Erkennung ebendieser nicht vorhanden, sodass keine korrekte Aussage getroffen werden kann.

Der Drop Layer zwingt das Model durch die Deaktivierung einzelner Neuronen dazu, andere Wege zu finden, um zur richtigen Lösung zu kommen, sodass auch andere Neuronen besser in der Lage sind, die Muster in den Daten zu erkennen.

```
Dropout(DROP),
```

Abbildung 21: Drop Layer in TensorFlow

Bei der Initiierung des Drop Layer Objekts in TensorFlow ist festzulegen, welcher Prozentsatz von Neuronen deaktiviert werden soll. Hierzu wurde der Hyperparameter DROP definiert, dessen Wert 0.2 besagt, dass dieser Prozentsatz 20% beträgt (Abbildung 21).

Nachdem nun die Hidden Layer, bestehend aus Konvolutions-, Pooling- sowie Drop Layern definiert wurden, können die Daten für die Entscheidungsfindung aufbereitet werden. Dies geschieht im Flatten Layer.

Flatten Layer

Der Flatten Layer dient als Vorbereitungslayer für das Dense Layer, welches den letztendlichen Output des Models ermittelt. Hierzu müssen die zuvor zweidimensional gehaltenen Daten in ein eindimensionales Format gebracht werden. Hierzu wird jedem Pixel innerhalb der Input Pools eine spezifische Position in der eindimensionalen Repräsentation zugewiesen wird (Abbildung 22).

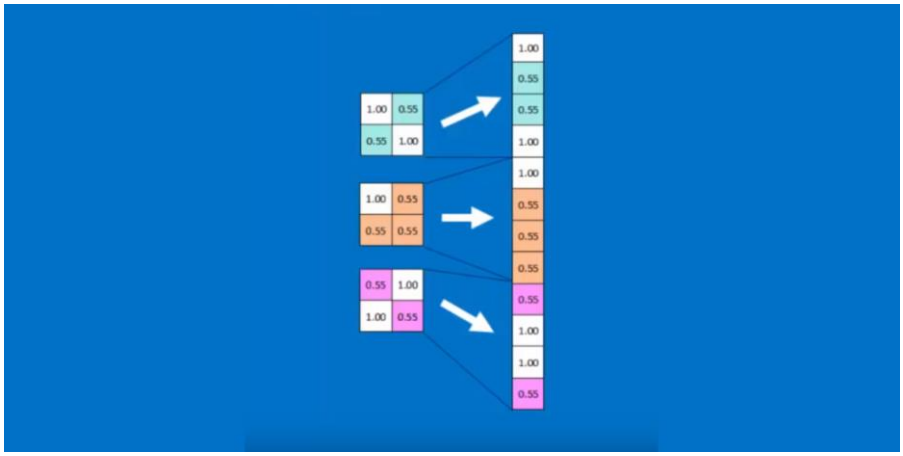


Abbildung 22: Umwandlung von 2D Daten in eindimensionale im Flatten Layer

```
Flatten(),
```

Abbildung 23: Flatten Layer in TensorFlow

Da der Flatten Layer lediglich eine Umwandlung durchführt, brauchen diesem bei der Initiierung durch TensorFlow keine besonderen Parameter übergeben werden (Abbildung 22).

Nachdem der Flatten Layer implementiert ist, muss zu allerletzt noch der Output Layer, der sogenannte Dense Layer implementiert werden.

Dense Layer

Das Dense Layer, auch Fully Connected Layer genannt, hat genau so viele Output Neuronen, wie es Entscheidungsmöglichkeiten gibt. Hierbei wird jeder Position des erwarteten Inputs eine Gewichtung zugewiesen, die dem Model Auskunft darüber gibt, wie wichtig ein Wert an dieser Stelle für einen gegebenen Output ist. Je größer der Einfluss eingeschätzt wird, desto höher fällt die Gewichtung aus. Aus dem tatsächlichen Input aus dem Flatten Layer wird dann entsprechend dieser Gewichtungen jede Position analysiert, sodass der Dense Layer am Ende für jeden möglichen Output einen Wert besitzt. Der mögliche Output mit dem höchsten Wert wird dann ausgegeben und repräsentiert die Entscheidung des gesamten Models.

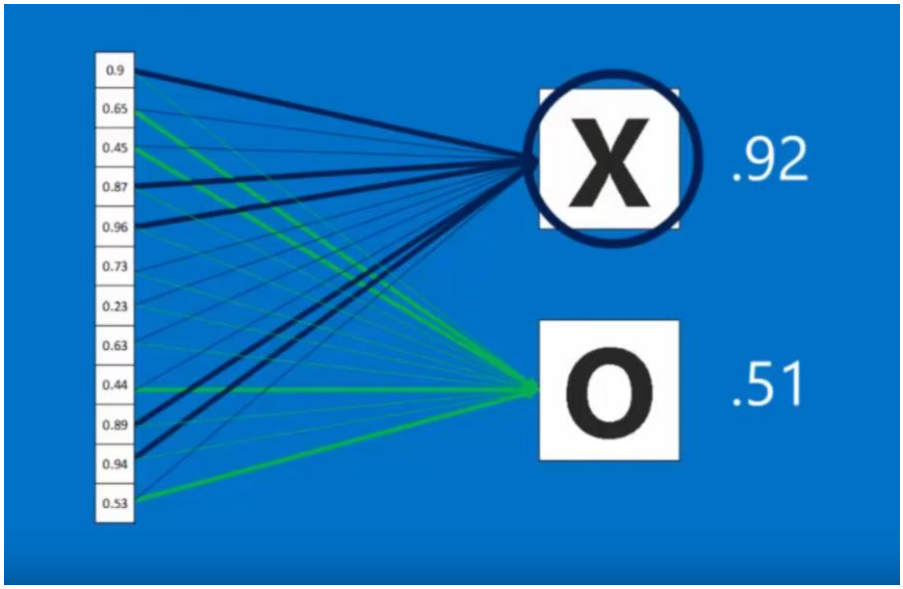


Abbildung 24: Entscheidungsfindung im Dense Layer

In der obigen Darstellung (Abbildung 23) repräsentieren die Werte, welche eine dicke Line zu einem Output haben eine starke Gewichtung in Richtung dieses Outputs. Wie sich erkennen lässt, entspricht der Wert, welche den jeweiligen Outputs zugewiesen wird in etwa dem Durchschnitt derjenigen Werte, denen eine besondere Relevanz zugesprochen wird. Da diese Werte nicht zwangsläufig Wahrscheinlichkeiten repräsentieren, ist es sinnvoll für die Interpretation dieser Daten wiederum eine Aktivierungsfunktion zu implementieren. Hierzu bietet sich die Aktivierungsfunktion „softmax“ an, welche die Vektorwerte aus dem Dense Layer (0,92 bzw. 0,51) in tatsächliche Wahrscheinlichkeiten umwandelt, mit denen weitergerechnet werden kann.

Bei der Implementation des Dense Layers mithilfe von TensorFlow sind bei der Initiierung des Objekts zusätzlich zu der Aktivierungsfunktion anzugeben, wie viele Outputmöglichkeiten es gibt. Diese Zahl entspricht der Anzahl der definierten Labels, in unserem Fall also zwei, für Katze und Hund (Abbildung 24).

```
Dense(2, activation='softmax'),
```

Abbildung 25: Implementation eines Dense Layers in TensorFlow

Modell bilden und kompilieren

Nachdem nun alle wichtigen Layerarten vorgestellt wurden, kann das eigentliche Modell definiert werden. Hierzu wird, wie bereits erwähnt, ein das Sequential Model der Keras Bibliothek verwendet, dass in Python durch eine Klasse repräsentiert wird, die bei ihrer Initiierung eine Liste an Layern als Parameter erwartet (Abbildung 26).

```
model = Sequential([
    Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same"),
    Dropout(DROP),
    Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same",activation="relu"),
    MaxPooling2D(pool_size=pool_size, padding="same"),
    Dropout(DROP),
    Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same",activation="relu"),
    Dropout(DROP),
    Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same",activation="relu"),
    MaxPooling2D(pool_size=pool_size, padding="same"),
    Dropout(DROP),
    Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same",activation="relu"),
    MaxPooling2D(pool_size=pool_size, padding="same"),
    Dropout(DROP),
    Conv2D(num_filters, filter_size, input_shape=(50,50, 1), padding="same"),
    MaxPooling2D(pool_size=pool_size, padding="same"),
    Flatten(),

    Dense(2, activation='softmax'),
])
```

Abbildung 26: Beispiel eines Convolutional Networks in TensorFlow

Bevor das Model trainiert werden kann, muss noch eine Optimierungsmethode festgelegt werden, nach der das Model die eigenen Gewichtungen anpasst. Anschließend kann das Model kompiliert werden. Hierzu müssen noch die statistische Methode zur Ermittlung des Loss Wertes (hierzu später mehr), sowie die zu überwachenden Metriken angegeben werden. Nach der Kompilierung müssen noch Meta Parameter, wie Dateinamen und Speicherort für die einzelnen Modelle, die während jedes Trainingsdurchlaufs generiert werden, festgelegt werden.

```
opt = tf.keras.optimizers.Adam(lr=LR, decay=DECAY)

model.compile(loss="sparse_categorical_crossentropy",
              optimizer=opt,
              metrics=["accuracy"]
              )

NAME = f"test-{int(time.time())}"
model.save_weights(f"Models/{NAME}-weights.h5")
tensorboard = TensorBoard(log_dir='Logs/{}/'.format(NAME))

filepath = f"num{num_filters}-size{filter_size}-pool{pool_size}-"+"{epoch:02d}-{val_acc:.3f}"
checkpoint = ModelCheckpoint("Models/{}.model".format(filepath, monitor='val_acc', verbose=1, save_best_only=True, mode='max'))
```

Abbildung 27: Kompilierung eines TensorFlow Models

Für das Modell dieses Fallbeispiels (Abbildung 27), wird die Optimierungsmethode „Adam“ verwendet, welche die derzeit gängigste Methode ist. Gleichzeitig wird angegeben, dass bei der Auswertung der Modellergebnisse der Accuracy Wert (hierzu später mehr) betrachtet werden soll. Anschließend wird das TensorBoard zur Auswertung des Trainingprozesses initiiert, indem diesem der Speicherort für die Logfiles mitgeteilt wird, sowie ein Checkpoint definiert, der nach jedem Trainingsdurchlauf das beste Modell unter dem angegebenen Dateinamen speichert. Dieser beinhaltet die Nummer des Trainingsdurchlaufs, die Filter- und Poolgrößen, sowie die Accuracy des Durchlaufs.

Nachdem jetzt die Trainings- und Validierungsdaten aufbereitet, alle Layer definiert, die Metaparameter festgelegt und das Model kompiliert wurden, kann das eigentliche Training des Models beginnen.

Training des neuronalen Netzes

Das Training des Models wird durch die Objektmethode „fit“ gestartet. Diese Methode erwartet als Parameter die Listen mit Trainings- und Validierungsdaten und die Anzahl der Durchläufe. Zusätzlich kann eine Liste an Callbacks übergeben werden, die festlegen, nach welchen Kriterien der interne Status und seine Statistiken gespeichert werden (Abbildung 28).

```
history = model.fit(
    X, Y,
    epochs=epoch,
    validation_data=(val_x,val_y),
    callbacks = [tensorboard, checkpoint]
)
```

Abbildung 28: Training des Modells mit TensorFlow

Die Variable *X* beinhaltet hierbei sämtliche Traningsbilder, während die Variable *Y* die Labels für die entsprechenden Bilder beinhaltet. Dasselbe gilt für die Variablen *val_x* und *val_y*, mit dem Unterschied, dass diese Listen die Validierungsdaten, nicht die Trainingsdaten beinhalten. Die Anzahl der Trainingsdruchläufe wird durch den Hyperparameter „epoch“ definiert, die „callbacks“ bestehen aus den soeben definierten TensorBoard und Checkpoint Objekten.

Kommentiert [VW2]:

Accuracy und Loss

Während des Trainings wird für jedes Bild ein Wert für Accuracy und einer für Loss ausgegeben. Am Ende jedes Epochs werden die Durchschnittswerte Werte für alle Trainings- und alle Validierungsdaten ausgegeben. Im Idealfall sind die korrespondierenden Werte aus Trainings- und Validierungsdaten gleich (Abbildung 29).

```
56448/60000 [===========================>..] - ETA: 0s - loss: 0.5044 - acc: 0.8443
56704/60000 [===========================>..] - ETA: 0s - loss: 0.5044 - acc: 0.8443
56960/60000 [===========================>..] - ETA: 0s - loss: 0.5044 - acc: 0.8442
57184/60000 [===========================>..] - ETA: 0s - loss: 0.5044 - acc: 0.8443
57408/60000 [===========================>..] - ETA: 0s - loss: 0.5045 - acc: 0.8442
57632/60000 [===========================>..] - ETA: 0s - loss: 0.5042 - acc: 0.8444
57888/60000 [===========================>..] - ETA: 0s - loss: 0.5038 - acc: 0.8445
58112/60000 [===========================>..] - ETA: 0s - loss: 0.5033 - acc: 0.8446
58336/60000 [===========================>..] - ETA: 0s - loss: 0.5038 - acc: 0.8446
58560/60000 [===========================>..] - ETA: 0s - loss: 0.5037 - acc: 0.8445
58816/60000 [===========================>..] - ETA: 0s - loss: 0.5033 - acc: 0.8447
59040/60000 [===========================>..] - ETA: 0s - loss: 0.5030 - acc: 0.8449
59264/60000 [===========================>..] - ETA: 0s - loss: 0.5032 - acc: 0.8448
59520/60000 [===========================>..] - ETA: 0s - loss: 0.5040 - acc: 0.8447
59776/60000 [===========================>..] - ETA: 0s - loss: 0.5036 - acc: 0.8448
60000/60000 [==============================] - 14s 241us/sample - loss: 0.5038 - acc: 0.8448 - val_loss: 0.1762 - val_acc: 0.9596
```

Abbildung 29: Accuracy und Loss während des Trainings mit TensorFlow

Der Accuracy Wert ist dabei eine prozentuelle Angabe darüber, wie viele der Daten korrekt klassifiziert wurden, während die Loss Angabe darüber Auskunft bietet, wie sicher sich das Model bezüglich der Entscheidungen ist. Der Loss Parameter ist dabei nicht prozentual angegeben und kann von Menschen nicht ohne weiteres interpretiert werden. Angestrebt wird ein Loss Wert von 0.

Eine einfachere Auswertung dieser Ergebnisse ermöglicht die Weboberfläche TensorBoard, welche die Logfiles des Modeltrainings auswerten und darstellen kann.

Auswertung des Trainings mithilfe von TensorBoard

Sobald das Training begonnen hat, kann auf der Weboberfläche des TensorBoards ermittelt werden, wie sich das Model im Laufe des Trainings entwickelt. Hierzu werden die Accuracy und Loss Werte für sowohl Trainings-, als auch Validierungsdaten aus jedem Epoch in einem Graphen dargestellt, um leichter zu ermitteln, ob sich die Werte so entwickeln wie erwartet (Abbildung 30).

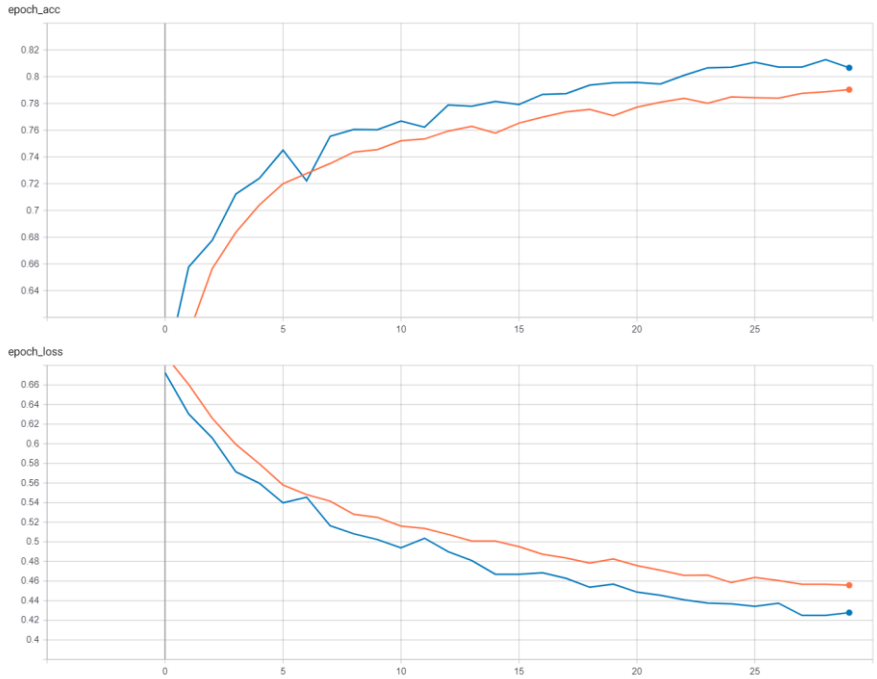


Abbildung 30: Loss und Accuracy dargestellt mithilfe von TensorBoard

Unterscheiden sich die Accuracy bzw. Loss Werte für Trainings- und Validierungsdaten stark voneinander, könnte dies ein Zeichen für Over- bzw. Underfitting sein und das Model, bzw. die Datengrundlage, müssen angepasst werden.

Overfitting und Underfitting

Im Falle des Overfittings hat das Model zu spezifische Muster gelernt hat und so Gemeinsamkeiten bei sich start unterscheidenden Daten nicht adäquat erkennen kann. Dies macht sich dadurch bemerkbar, dass die Accuracy bei den Validierungsdaten rapide abnimmt, während die Accuracy bei den Testdaten nach wie vor steigt (Abbildung 31).

Abbildung 31: Beispiel für Overfitting dargestellt mithilfe von TensorBoard

Je mehr Layer und Neuronen vorhanden sind, desto höher ist die Chance, dass Overfitting auftritt, da durch ansteigende Anzahl von Neuronen immer spezifischere Muster erkannt werden können. Wird ein Overfitting erkannt, sollte das Training abgebrochen werden und das Model angepasst werden. Hierbei empfiehlt sich insbesondere das hinzufügen von weiteren Drop Layern.

Underfitting hingegen bedeutet, dass das Model zu allgemeine Muster gelernt hat, die zu wenig Auskunft über das Erkenntnisobjekt liefern. Underfitting lässt sich daran erkennen, dass die Accuracy und Loss Werte der Validierungsdaten signifikant schwächer sind als die der Trainingsdaten (Abbildung 31).

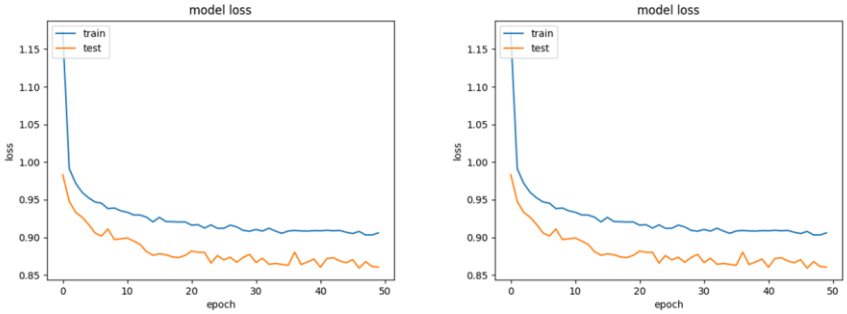


Abbildung 32: Beisiel für Underfitting dargestellt mithilfe von TensorBoard

Tritt Underfitting auf, kann dies verschiedene Gründe haben. Zum einen könnte die Anzahl an Layern und Neuronen zu gering sein, um spezifische Muster erkennen zu können, andererseits könnte aber auch die Datengrundlage zu homogen sein, sodass gewisse relevante Muster in den Daten nicht auftreten. In beiden Fällen sollte auch im Falle des Underfitting das Training abgebrochen werden und entweder das Model, oder die Datengrundlage angepasst werden.

Nutzung des neuronalen Netzes

Die Nutzung des Models erfolgt durch einlesen einer der, während des Training generierten, Modeldateien mithilfe der Funktion „load_model()“ (Abbildung 33).

```
path_to_model="" #Pfad zum Model angeben
model = tf.keras.models.load_model(path_to_model)
```

Abbildung 33: Laden eines Models mit TensorFlow

Hierbei sollten die einzelnen Modelle anhand ihrer Accuracy und Loss Werte in den Validierungsdaten ausgewertet werden und dasjenige Modell geladen werden, welches die größte Accuracy bei einem möglichst geringen Loss Wert ausgewählt werden.

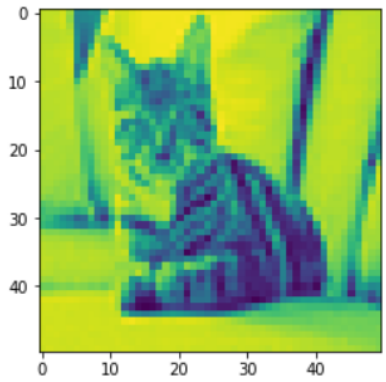
Nun kann ein Beispielbild, welches entweder einen Hund oder eine Katze darstellt, aus dem Internet heruntergeladen werden. Soll dieses Bild von dem Model klassifiziert werden, muss das Bild entsprechend der Trainings- und Validierungsdaten formatiert werden. Das daraus resultierende Numpy Array, muss der *„predict“* Methode des eingelesenen Model Objekts als Parameter übergeben werden.

Der Output der „predict“ Methode ist eine Liste, die genauso viele Elemente beinhaltet, wie im Dense Layer als Output definiert wurden, also genau so viele Elemente, wie es Labels gibt (Abbildung 34).

```
path_to_img = "katze2.jpg" #Pfad zum Bild angeben
img = Image.open(path_to_img).convert("I")
img.load()
img = img.resize((50,50), Image.LANCZOS)
data = np.asarray( img, dtype="int32" )

plt.imshow(data)
plt.show()

data = data.reshape((50,50,1))
data = data/255
print(data.shape)
```




```
(50, 50, 1)
```

```
prediction = model.predict([[data]])
print(prediction)
```

```
[[0.9497715  0.05022851]]
```

Abbildung 34: Nutzung des trainierten Models

Bei der Verwendung unseres Modells werden also entsprechend der Labels Katze und Hund zwei verschiedene Werte ausgegeben. Dabei entspricht der Wert an der Listenposition [0] dem Wert für Katze, da bei der numerischen Definition der Labels dem Label „Cat“ die Zahl 0 zugeordnet wurde. Der Wert an der Listenposition [1] ist im Umkehrschluss der Wert für einen Hund.

Dank der Aktivierungsfunktion „softmax“, welche im Dense Layer ausgewählt wurde, repräsentieren diese Werte Wahrscheinlichkeiten. In dem obigen Fall ist sich das Model demnach zu ~95% sicher, dass es sich bei dem Bild um eine Katze handelt, während das Bild mit einer Wahrscheinlichkeit ~5% einen Hund darstellt.

Abschluss

Anhand dieses Ratgebers wurde hoffentlich deutlich, wie ein Konvolutionsnetz aufgebaut ist und arbeitet, sowie ein solches Netz mithilfe des Frameworks TensorFlow selbst implementiert und genutzt werden kann. Ein weiteres beliebtes Beispiel für die Implementation eines solchen Netzes ist das sogenannte MNIST Modell, welches handgeschriebene Ziffern im Bereich von 0 bis 9 erkennen soll und auch als „Hello World“ des maschinellen Lernens bezeichnet wird.

Dieses Papier sollte einen Überblick über die Konzepte und Schritte, die zur eigenen Implementation des MNIST Projekts notwendig sind, aufgezeigt haben.

Alle Codebeispiele, sowie die Musterlösungen für das *„Cats vs. Dogs“* und *MNIST* Projekt sind für Mitglieder des Studiengangs Wirtschaftsinformatik an der Hochschule Heilbronn im GitLab einsehbar. Der Link zum Repository, in welchem auch diese Ausarbeitung zu finden ist, befindet sich im Anhang dieses Papiers.

Anhang

URL zum GitLab Projekt für WIN Studenten der HHN:

https://gitlab.win.hs-heilbronn.de/vweber/fsew-ws19-tensorflow