Projektarbeit
Neuronales Netz zur Bilderkennung mit dem CIFAR-10 Datensatz

Nachvollziehbare Schritte

Georg Aubele, 13.06.23

1 Kurze Darstellung des Problembereichs / Aufriss des Themas

1.1 Inhaltlich

In dieser Arbeit werden – nach dem Einlesen und Aufbereiten der Daten – einfache neuronale Netze bezüglich ihrer Möglichkeiten zur Bilderkennung im CIFAR-10 Datensatz untersucht. Am Ende wird jeweils die beste Version eines jeden Netzes dazu verwendet, um anhand von selbst erstellten Bildern (die ebenfalls entsprechend aufbereitet werden mussten) die Kategorisierung zu testen.

Ziele dieser Arbeit

1. Einlesen und Aufbereiten der Daten für die weitere Verwendung in neuronalen Netzen
2. Erstellung verschiedener neuronaler Netze und Verbesserung der Erkennungsgenauigkeit in den Netzen durch Veränderung der jeweiligen Parameter
3. Versuch der Kategorisierung von selbst erstellten Bildern

Quellen

-/-

1.2 Begründung des Themas

Darstellung der Relevanz des Themas

Objekterkennung auf Bildern bzw. in Videos[1] ist ein zentrales Thema bei autonomen Prozessen[2]. Deshalb muss und wird diese Erkennung immer weiter vorangetrieben werden. Ziele der Weiterentwickung sollten mMn sein, dass man zum einen die Erkennungsgenauigkeit steigert, zum anderen die Komplexität der Netze möglichst gering hält, um ein System bauen zu können, das mit begrenzten Ressourcen dennoch schnell und flüssig reagieren kann[3].

Darstellung eines persönlichen Erkenntnisinteresses

Im Hinblick auf eine zukünftige Arbeitsstelle kann es sicher nicht schaden, sich auch mit der Objekterkennung auseinandergesetzt zu haben.
Zusätzlich ist es hierbei natürlich auch notwendig, die entsprechenden neuronalen Netze zu optimieren. Da es hierzu keine wirklichen „Regeln“ oder klare Algorithmen gibt, ist eine Beschäftigung mit so einem Thema auch deswegen interessant, um ein Gefühl für die einzelnen Stellschrauben an den neuronalen Netzen zu entwickeln. Dieses bekommt man nur, wenn man sich aktiv mit der Optimierung beschäftigt.
Ein Projekt, das ich vielleicht einmal realisieren möchte, ist die Verfolgung unserer Zwergkaninchen mit Hilfe einer Webcam. Dazu müssen die Kaninchen zumindest als solche er-

1 Im Endeffekt eine Aneinanderreihung von Bildern
2 Autonomes Fahren von Autos, automatischer Einlass an Sicherheitsschranken für bekannte Personen, Erkennen und persönliche Ansprache von bekannten Personen durch Service-Roboter, Verkehrsüberwachung, Automatisierung im Haushalt ...
3 Auf Geschwindigkeit kommt es besonders im Bereich des autonomen Fahrens an, da hier die Zeit eine kritische Rolle spielt. Gleichzeitig ist bei der aktuellen Batterie-Situation ein System gefordert, das nicht zu viel Energie benötigt – auch hier ist Einfachheit zu priorisieren.

kannt werden. Eine Unterscheidung der beiden wäre dann noch ein „nice to have“. Aber das ist im Moment Zukunftsmusik.

2 Nachvollziehbare Schritte

2.1 Der Datensatz

Der CIFAR-10 Datensatz[4] ist ein bekannter Datensatz bestehend aus 60000 Farbbildern (drei Farbkanäle – Rot, Grün, Blau), gleichverteilt aus den 10 Kategorien

0 airplane
1 automobile
2 bird
3 cat
4 deer
5 dog
6 frog
7 horse
8 ship
9 truck

Davon werden 50000 Bilder für das Training der Netze verwendet und 10000 zum Testen. Im Testpaket sind von jeder Kategorie genau 10000 enthalten.
Bei den Kategorien „automobile“ und „truck“ wurde darauf geachtet, dass es keine „Zweifelsfälle“ gibt, es wurden keine Pickups oder kleine LKWs mit aufgenommen, ein „truck“ ist immer ein großer LKW, bei „automobile“ wurden keine Fahrzeuge mit Ladefläche verwendet.
Der Trainingsdatensatz liegt in 5 Dateien vor, in denen jeweils 10000 Bilder enthalten sind. Der Testdatensatz ist ein einzige Datei mit 10000 Bildern.

Die Dateistruktur der Trainings- und Testdateien ist ein Dictionary, in dem jeweils ein Schlüssel und ein dazugehöriger Wert vorhanden sind:

Schlüssel	Wert
data	Liste aus 10000 Einträgen; jedes Element ist eine Liste aus 3072 Integer, die ersten 1024 Zahlen entsprechen den Rot-Werten die nächsten 1024 den Grün-Werten und die letzten den Blauwerten. Alle Werte haben den Datentyp einer 8-bit-Integerzahl ohne Vorzeichen.[5] Das Bild selbst hat eine Auflösung von 32 mal 32 Pixel, das entspricht den 1024 Einzelwerten im jeweiligen Farbkanal.
labels	Die jeweilige Kategorie des entsprechenden Bildes als Zahl zwischen 0 und 9 (siehe oben)
batch_label	Beschreibung des Datensatzes
file_names	Dateinamen der einzelnen Bilder

4 Quelle: https://www.cs.toronto.edu/~kriz/cifar.html
5 Was genau den 256 Farbabstufungen innerhalb einer Farbe entspricht.

Für die Verarbeitung sind nur die ersten beiden Zeilen wichtig.

Der Datensatz gilt als Benchmark-Datensatz für die Evaluierung von Bildklassifizierungsalgorithmen. Aufgrund der geringen Größe der Bilder und der Vielfalt der Klassen stellt der Datensatz schon eine Herausforderung dar[6].

2.2 Fragestellung und Ansatz

2.2.1 Einlesen und Aufbereiten der Daten für die weitere Verwendung in neuronalen Netzen

Jedes Bild liegt als Abfolge von 3072 Zahlen in einer Liste vor[7], diese muss eingelesen und für die Verwendung im neuronalen Netz die eine passende Form gebracht werden.

Die Fragestellung ist:
Ist ein Einlesen und Aufbereiten der Bilder zur Weiterverwendung ohne großen Aufwand möglich?

2.2.2 Erstellung verschiedener neuronaler Netze und Verbesserung der Erkennungsgenauigkeit in den Netzen durch Veränderung der jeweiligen Parameter

Es gibt viele verschiedene Strukturen von neuronalen Netzen mit verschieden Arten von Schichten, jeweils mit mehreren Parametern, die die Effizienz und den Rechenaufwand bei der Verwendung der Netze stark beeinflussen. Dabei ist nicht nur darauf zu achten, dass die antrainierte Genauigkeit beim Erkennen der Bilder hoch ist (Trainingsdatensatz), sondern viel wichtiger ist die korrekte Kategorisierung von Bildern mit denen nicht trainiert wurde (Testdatensatz).

Die Fragestellung ist:
Können relativ einfache neuronale Netze aufgebaut und so abgestimmt werden, dass die Erkennungsrate mit dem Testdatensatz einen akzeptablen Wert erreicht?

2.2.3 Versuch der Kategorisierung von selbst erstellten Bildern

Die Anwendung in „freier Wildbahn“ ist mir ein Anliegen, denn kein System ist wirklich brauchbar, wenn es seine Tauglichkeit nur unter Laborbedingungen bewiesen hat. Die einfachste Möglichkeit dafür ist, selbst gemachte Fotos in das entsprechende Format zu bringen und diese dann mit Hilfe der Netze einsortieren zu lassen. Dabei werde ich die Qualität der Bilder (Kontrast und Größe des Objekts) variieren und auch versuchen, eines meiner Kaninchen kategorisieren zu lassen[8].

6 Bei der Beschreibung der Schritte werde ich später noch einige Bilder ausgeben, es sind wirklich nicht sehr viele Konturen und Details zu erkennen, was die Objekterkennung sehr erschwert!
7 Diese Listen sind wiederum Elemente einer Liste im Dictionary.
8 Auf die Kategorie bin ich jetzt schon gespannt.

Die Fragestellung ist:
Können eigens erstellte Bilder mit wenig Aufwand in die notwendige Form gebracht und dann auch korrekt eingestuft werden? Wie reagieren die Netze auf Objekte, die zu keiner Kategorie passen?

2.3 Methode und Ergebnisse[9]

2.3.1 Einlesen und Aufbereiten der Daten

Die Dateien mit den Trainings- bzw. Testdaten lagen jeweils als Binärdateien vor. Das Einlesen geschah mittels der pickle-Bibliothek, das OS-Paket ermöglicht, Dateipfade und Dateinamen zu verknüpfen.

2.3.1.1 Variablen und Funktionen

Zunächst werden einige Variablen definiert, die das Einlesen der Dateien erleichtern sollten, darunter der Dateipfad und die Dateinamen. Außerdem wird eine Liste der Dateien angelegt, um das Einlesen in einer for-Schleife realisieren zu können.

```
# %% Daten importieren

folder = "data"

file_1 = "data_batch_1"
file_2 = "data_batch_2"
file_3 = "data_batch_3"
file_4 = "data_batch_4"
file_5 = "data_batch_5"
file_test = "test_batch"
file_meta = "batches.meta"

train_file_list =[file_1, file_2, file_3, file_4, file_5]
```

Für das eigentliche Einlesen und das exemplarische Darstellen einiger Bilder werden Funktionen definiert:

9 Ich habe mich entschlossen, beides zusammen zu erläutern, da die Ergebnisse jeweils auch Einfluss auf die weitere Vorgehensweise hatten.

```python
def unpickle(folder, file):
    """ liest die Datei ein und extragiert das Dict"""
    file_path = os.path.join(folder, file)
    with open(file_path, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict


def show_img(dict, i):
    """"zeigt das i-te Bild aus dem dictionary dict"""
    pic = np.array(dict[b"data"][i])
    pic = pic.reshape((3,32,32))
    pic = pic.transpose(1,2,0)

    # Bild anzeigen
    plt.imshow(pic)
    plt.show()


def dicts_2_arrs(dict):
    """x-vals werden in ein numpy-Array der entsprechenden Form gebracht,
    die Werte selbst in float umgewandelt und normalisiert
    die y-vals werden in ein array geschrieben"""
    x_vals = np.array(dict[b"data"]).reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)
    x_vals = x_vals.astype(np.float32)
    x_vals /= 255.0
    y_vals = to_categorical(np.array(dict[b"labels"]))
    return x_vals, y_vals


def cats_2_vecs(dict):
    """Die Kategoerien der Bilder werden ausgelesen und in ein Array aus
    Einheitsvektoren umgeschrieben"""
    return to_categorical(dict[b"labels"])
```

Die Funktion unpickle liest die Datei aus dem entsprechenden Pfad und gibt das darin enthaltene Dictionary zurück.

Da Funktionen möglichst nur eine Aufgabe erfüllen sollten, wird der entsprechende Eintrag im Dictionary in der Funktion dicts_2_arrs dann weiter in ein Array umgewandelt und zurückgegeben. Beim Schlüssel steht ein b vor den Anführungszeichen, da es sich um einen binären String handelt. Dies ist hier wichtig, da der Schlüssel sonst nicht gefunden wird.
Nach einem Reshape (die Daten werden aus der Liste in die korrekte, dreidimensionale Form gebracht) und einem transpose (die Dimensionen müssen vertauscht werden – vergleichbar mit einem Auseinanderschneiden verschiedener Spalten unterschiedlicher Tabellen und dem Zusammenkleben in einer anderen Reihenfolge) sind die Daten in der für das Netzwerk passenden Form.

Die Funktion cats_2_vecs wandelt die Kategoriebezeichnungen der einzelnen Bilder in jeweils einen Einheitsvektor um: Jede Zahl wird in ein Tupel aus 10 Zahlen umgewandelt, bei dem dann an der entsprechenden Stelle (wenn die Zahl eine 0 war an der 0. Stelle, wenn die Zahl eine 1 war, an der 1. Stelle usw.) eine 1 steht. Dies wird benötigt, da die Netze am Ende jeweils eine Ausgabe-Schicht mit 10 Knoten haben und für jeden Knoten eine Wahrscheinlichkeit (zwischen 0 und 1) angeben, ob das in das Netz eingespeiste Bild in die jeweilige Kategorie gehört. Dies wird dann mit dem entsprechenden Vektor verglichen.

Die Funktion show_img nimmt ein Dictionary auf und zeigt einen Eintrag in der entsprechenden Liste als Bild an, nachdem die Daten in die für die Anzeige korrekte Form gebracht wurden.

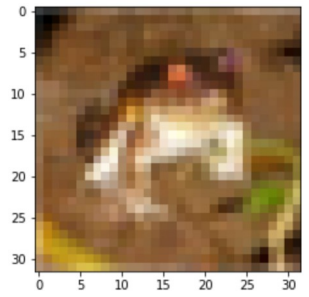
2.3.1.2 Umwandeln der Daten und Anzeigen der Bilder

Nun folgt das eigentliche Einlesen der Dateien und eine exemplarische Ausgabe von ein paar Bildern:

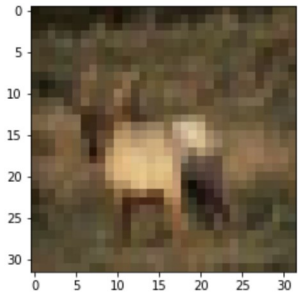
```
# Liste der Kategorien mit entsprechenden Ziffern
cat_dir = unpickle(folder,file_meta)
cat_dir[b"num_vis"]
cat_list = [s.decode("utf8") for s in cat_dir[b"label_names"]]
for num, val in enumerate(cat_list):
    print(num, val)


batch_1_dict = unpickle(folder, file_1)
batch_1_dict[b"data"]
batch_1_dict[b"data"][0]

batch_1_target = cats_2_vecs(batch_1_dict)


# Ein paar Bilder anzeigen
for i in range(10):
    show_img(batch_1_dict, i)
```

Im ersten Teil werden die Kategorien eingelesen und in der Konsole ausgegeben, diese sind oben unter 2.1 zu sehen. Das „b“ vor dem eigentlichen String ist wieder notwendig, da der String in der Datei in einem binären Format vorliegt.
Auch werden die Bezeichnungen vorher in ein lesbares Format (utf8) umgewandelt.

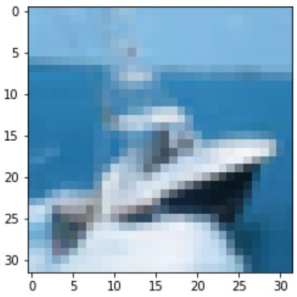
Als nächstes wird die erste Trainingsdatei probehalber eingelesen und einige Bilder vom Anfang des Datensatzes ausgegeben.
Hier einige Beispielbilder:



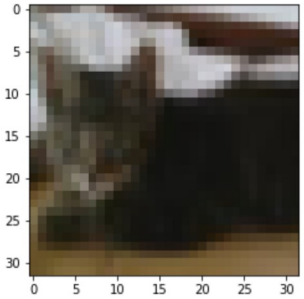
Ein Pfeiffrosch



Ein Hirsch



Ein Boot



Eine Katze

Ich denke, an diesen Beispielen wird klar, dass hier eine Objekterkennung und -unterscheidung in Kategorien schon eine Herausforderung darstellt!

Da das Einlesen der ersten Datei problemlos funktioniert hat, kann nun mit der Verarbeitung aller Trainingsdateien weitergemacht werden:

```
# %% Trainingsdaten alle in zwei Arrays (trainX und trainy) packen

shape_X = (1,32,32,3)
shape_Y = (1,10)

train_X = np.empty(shape_X, dtype=np.float32)
train_Y = np.empty(shape_Y, dtype=np.uint8)

for file in tqdm.tqdm(train_file_list):
    dict = unpickle(folder, file)
    x, y = dicts_2_arrs(dict)
    train_X = np.concatenate((train_X, x), axis=0)
    train_Y = np.concatenate((train_Y, y), axis=0)

train_X = train_X[1:50001]
train_Y = train_Y[1:50001]
```

Die Shapes sind die Formen, in denen die Daten als x-(Eingabe) und y-Werte (Ausgabe) vorliegen müssen. Damit wird dann jeweils ein leeres Numpy-Array erstellt[10] und in einer for-Schleife gefüllt Das Füllen erfolgt mit dem Aneinanderreihen der Listen in das Array. Das leere Anfangselement muss am Schluss wieder entfernt werden.
Auf die gleiche Weise werden nun auch die Testdaten eingelesen und umgeformt, hier ist allerdings keine Schleife notwendig, da es sich nur um eine Datei handelt.

```
# %% Testdaten ebenfalls in arrays umandeln

test_dict = unpickle(folder, file_test)
test_X, test_Y = dicts_2_arrs(test_dict)
```

Somit sind die Daten in der gewünschten Form im Speicher vorhanden.

Nebenbemerkung:
Der CIFAR-10 Datensatz liegt auch in keras als Datensatz vor, es wäre auch möglich gewesen, ihn „einfach“ einzulesen.

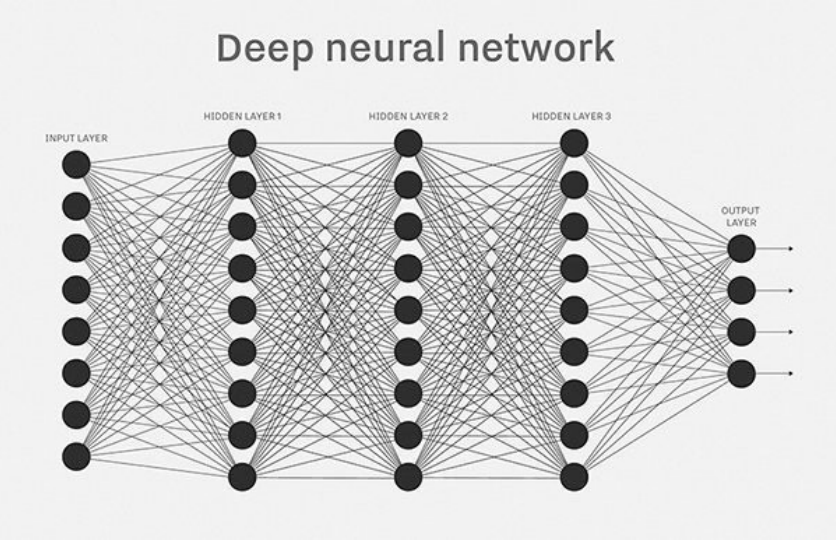
10 Dies ist in Numpy leider etwas umständlich.

```
Zum Einlesen wäre auch folgendes möglich:
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    assert x_train.shape == (50000, 32, 32, 3)
    assert x_test.shape == (10000, 32, 32, 3)
    assert y_train.shape == (50000, 1)
    assert y_test.shape == (10000, 1)
```

Ich nehme aber das Einlesen und Vorbereiten von Hand vor, um auch darin Übung zu bekommen.

2.3.2 Erstellung verschiedener neuronaler Netze und Verbesserung der Erkennungsgenauigkeit in den Netzen durch Veränderung der jeweiligen Parameter

2.3.2.1 Kurzer Überblick über den Einfluss von Hyperparametern



In einem neuronalen Netz[11] werden die Eingangsdaten durch mehrere Schichten geschickt. In den Schichten (layers) gibt es einzelne Knoten (nodes), von denen ein Signal einer bestimmten Stärke an die Knoten der nächsten Schicht weitergegeben wird.
Die Stärke des Signals wird durch Gewichte (weights) geregelt. Die Aktivierungsfunktion (activation) verändert dieses Signal noch leicht nach einem vorgegebenen Algorithmus, um eine Nicht-Linearität einfließen zu lassen. Diese ist wichtig, um nicht-lineare Zusammenhänge auffinden zu können.
Bei einem Trainingsdurchgang (train) wird am Ende das durch das Netz gefundene Ergebnis (predict) mit dem tatsächlichen (target) verglichen. Je nach Ähnlichkeit der beiden wird ein Fehler (loss) berechnet. Theoretisch werden dann mit einem „back propagation“ genannten Prozess die Gewichte im Netz rückwärts laufend entsprechend der Differenz von target und predict in den einzelnen Schichten angepasst. Praktisch wird der Fehler über mehrere Durchgänge aufaddiert und dann eine back propagation veranlasst (siehe „Batches“).

11 Quelle des Bildes: https://cdn.ttgtmedia.com/rms/onlineImages/deep_neural_network_desktop.jpg

Die Optimierungsfunktion (optimizer) bestimmt zusammen mit dem loss, nach welchem Schema und in welchem Umfang die Gewichte angepasst werden sollen.
Die Lernrate (learning rate) gibt noch einen Faktor an, wie stark die Gewichte bei jedem Durchlauf angepasst werden. Ist die Lernrate gering, geht der Lernprozess langsamer, dafür kann es sein, dass bessere Ergebnisse erzielt werden, da es – grob ausgedrückt – dabei darum geht, möglichst das globale Minimum[12] einer Funktion zu finden. Eine zu kleine Lernrate kann den Lernerfolg aber auch behindern. Bei zu großer Lernrate kann es ebenfalls sein, dass man die Minima verfehlt, weil sozusagen die Schrittweite zu groß ist.
Die Trainingsdaten werden in zufällig zusammengestellte Teile (Batches) aufgeteilt. Diese Einteilung hat mehrere Vorteile: Zum einen können bei manchen Systemen Prozesse parallel berechnet werden, was dann den Vorgang beschleunigt. Bei sehr großen Datensätzen muss mit der Unterteilung auch nicht immer der komplette Datensatz im Speicher gehalten werden. Am wichtigsten ist wohl, dass nach dem Durchlauf eines Teils eine Gewichtsanpassung durchgeführt wird. Sind die Batches zu klein, verringert sich der Lernerfolg, da zu wenig Daten zum Lernen je Batch vorhanden sind. Sind die Batches zu groß, wird die Anpassung seltener ausgeführt, auch hier ist der Lernerfolg dann fraglich.
Die optimale Batch-Größe ist stark vom Problem und vom Datensatz abhängig[13]. Ist der gesamte Trainingssatz durchgelaufen, ist eine Epoche beendet. Die Anzahl der Epochen gibt an, wie oft der gesamte Trainingssatz das Netzwerk durchlaufen soll.

2.3.2.2 Die verschiedenen Arten von Schichten

Die von mir verwendeten Netze haben die folgenden Arten von Schichten:

- Convolutional Layer: Diese Schicht dient zur Erkennung von Kanten, Strukturen oder anderen Mustern. Es wird eine Matrix der Größe kernel_size[14] über das Bild geschoben (strides gibt dafür die Schrittweite an). Die eigentliche Faltung (Convolution) ist ein mathematischer Vorgang, bei dem die Gewichte der Matrix mit den jeweiligen Eingangssignalen verrechnet werden. Je mehr kernels (oder filters) man angibt, desto öfter wird eine solche Faltung ausgeführt. Da die Faltung die Matrix eigentlich verkleinert, kann man mit Hilfe des padding-Parameters angeben, ob man eine Verkleinerung zulassen („valid“) oder verhindern („same“) möchte. Verhindert man die Verkleinerung, so hat es den Vorteil, dass Strukturen an den Rändern der Bilder eine ähnliche Aufmerksamkeit bekommen wie die Strukturen im Innern. Nachteil ist ein höherer Rechenaufwand.
- MaxPooling Layer: Diese Schicht dient der Reduktion des Bildes auf „Wesentliches“. Auch hier wird eine Matrix über das Bild geschoben, der jeweils größte Wert, der Einträge unter der Matrix wird als Ausgabe weitergegeben. So wird z.B. ein 3x3-Bereich auf eine Zahl reduziert.
- Dropout Layer: Diese ist keine Schicht im engeren Sinne[15], sie stellt nur eine Vorschrift dar. Hier wird nach Zufall ein bestimmter Anteil (rate) der Nodes ausgewählt und „stumm geschaltet“. Dies erhöht beim Training die Robustheit des Netzes und verringert den Effekt des Overfittings[16].

12 Oder zumindest ein möglichst tiefes lokales Minimum
13 Für meine Arbeit hat sich in Vortests mit mehreren Netzen 100 als gut herausgestellt.
14 Bei Angabe von nur einer Zahl ist diese quadratisch.
15 Genau wie die Activation Layer
16 Zu starke Anpassung des Netzes an seine Trainingsdaten. Dies verschlechtert die Erkennung von neuen, unbekannten Daten.

- Dense Layer: Die „normale“ Schicht. Hier sind alle Nodes mit allen Nodes der vorgehenden und nachfolgenden Schichten über Gewichte verbunden. Es greifen keine weiteren besonderen Algorithmen ein.

2.3.2.3 Vorbereitungen für die Erstellung der Netzwerke

Damit später das Verbessern der Netzwerke einfacher von Statten geht, werden die Parameter und Hyperparameter hier zu Beginn zusammengefasst:

```
# %% CNN aufbauen - Parameter

width = 32
height = 32
depth = 3

num_classes = 10

train_size, test_size = train_X.shape[0] , test_X.shape[0]

# %%# Hyperparameter
epochs = 30
batch_size = 100
lr = 0.003
# optimizer = optimizers.Adam(learning_rate = lr)
optimizer = optimizers.Adam() # default lr = 0.001
```

Die Parameter width, height und depth beziehen sich auf die Bilder, wobei die depth die Farbtiefe von 3 Farbkanälen (Rot, Grün, Blau) darstellt. num_classes stellt die Anzahl der Kategorien dar und train_size und test_size gibt jeweils die Anzahl der Trainings- bzw. Testitems wieder.
Die Bedeutung der einzelnen Hyperparameter wurde bereits oben beschrieben.

- Epochs ist die Anzahl der Durchläufe durch den gesamten Trainingsdatensatz
- batch_size gibt die Größe eines Teils an, in die der Datensatz während einer Epoche unterteilt wird
- lr steht für die Lernrate des Netzes
- optimizer ist der verwendete Optimierungsalgorithmus im Lernprozess. Adam hat sich hier in den meisten Fällen bewährt. Gibt man innerhalb der Klammer keine Lernrate an, so verwendet Adam laut Dokumentation automatisch 0.001.

Der Code für das Training schaut folgendermaßen aus:

```
# ================================================================================
# TRAINING
# ================================================================================

history = model.fit(x=train_X, y=train_Y, verbose=1, batch_size=batch_size,
                    epochs=epochs, validation_data=[test_X, test_Y])
```

Das Fitting wird in einer Variable „history“ abgespeichert, damit man später die Trainingswerte abrufen kann. Verbose ist ein Parameter, mit dem man die Ausgabe während des Trainings steuern kann[17].

17 0 bedeutet keine Ausgabe, 1 bedeutet einen ständig aktualisierten Fortschrittsbalken für jede Epoche, 2 ergibt eine Ausgabe pro Epoche

Ein Abspeichern und Laden des trainierten Netzes hat den Vorteil, dass man – bei Neustart des Rechners, notwendigem Wechseln der virtuellen Umgebung etc. – diese später nicht noch einmal neu trainieren muss.

Der Code dafür ist der folgende:

```
# %% Speichern des Modells

model.save(model_name)

# %% Laden eines Modells

# model_name = "M1"
# model_name = "M2"
model_name = "M3"

model = models.load_model(model_name)

print(model.summary())
```

Der Dateiname ist der Name des benutzten Netzes. Da dieser bei der Aktivierung des Netzes automatisch gesetzt wird, kann man nach dem Training dieses einfach speichern.
Zum Laden eines bestimmten Netzes wird der entsprechende Name aktiviert, die anderen auskommentiert.
Nach dem Laden werden die Daten des Netzes in der Konsole zur Überprüfung ausgegeben.

Am Ende eines jeden Trainingsvorgangs möchte ich die Daten grafisch darstellen. Durch das vorherige Abspeichern in der history-Variable stehen die Daten pro Epoche für einen Plot zur Verfügung.
Hier der dazugehörige Code:

```python
# %% Plots

# Zugriff auf die Accuracy-Werte
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
train_loss = history.history['loss']
val_loss = history.history['val_loss']

# Plot erstellen
fig, axs = plt.subplots(1, 2)

# Plot der Accuracy-Werte
axs[0].plot(train_accuracy, label='Train Accuracy')
axs[0].plot(val_accuracy, label='Validation Accuracy')
# axs[0].set_ylim(0.94, 1)
axs[0].set_xlabel('Epochen')
axs[0].set_title('Accuracy')

axs[0].legend()


axs[1].plot(train_loss, label='Train Loss')
axs[1].plot(val_loss, label='Validation Loss')
# axs[1].set_ylim(0, 0.2)
axs[1].set_xlabel('Epochen')
axs[1].set_title('Loss')
axs[1].legend()

fig.suptitle("M: " + model_name + ", Eps: " + str(epochs) +", Bt : " + str(batch_size) + ", LR: "+ str(lr))
plt.tight_layout()
plt.show()

graph_folder = ".//graph//"
graph_name = str(img_counter).zfill(2)+" M"+model_name + " E" + str(epochs) +" Bt" + str(batch_size) + " L" + str(lr) +".png"

graph_file = os.path.join(graph_folder, graph_name)

fig.savefig(graph_file, dpi=600)
img_counter += 1
```

Die entsprechenden Werte werden aus der history ausgelesen und ein Variablen gepackt, dann die Plots vorbereitet (zwei Graphen nebeneinander), indem die Achsen und Graphen beschriftet und jeweils eine passende Überschrift vergeben werden. Am Ende werden die Graphen angezeigt und abgespeichert.
Der image_counter wird um 1 erhöht, damit das nächste Bild beim Abspeichern die passende durchlaufende Nummer hat.

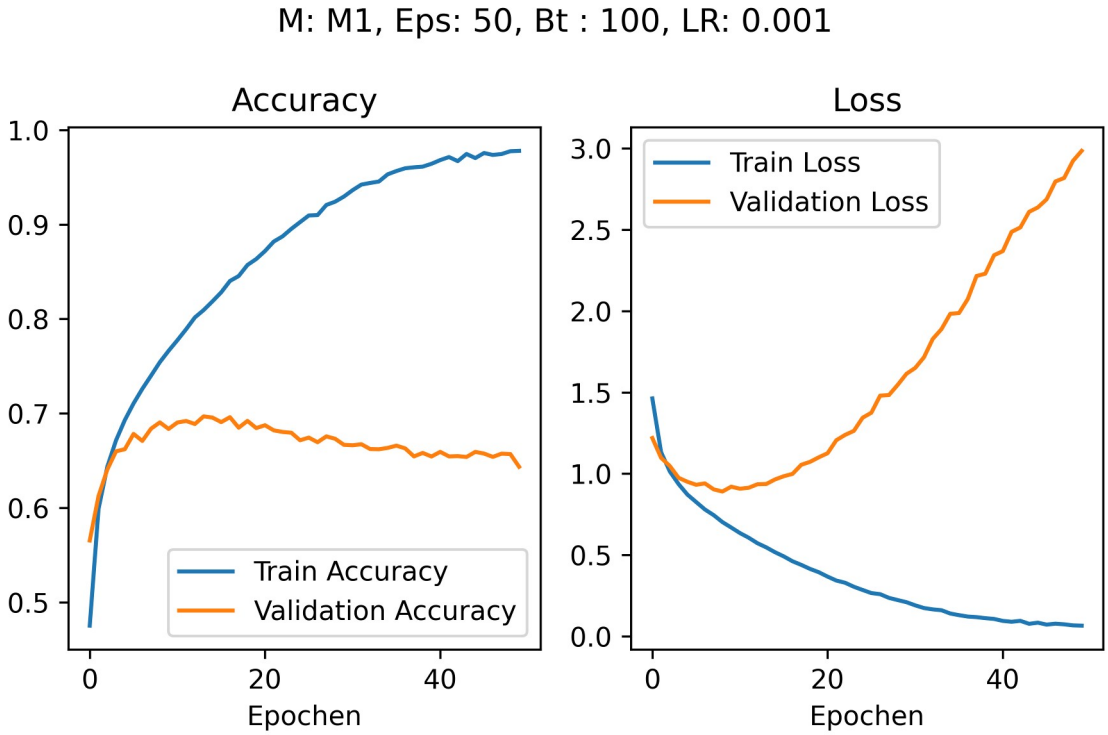
2.3.2.4 Das erste Netz

```
# %% Das eigentliche Model 1

model_name = "M1"

model = models.Sequential()

model.add(layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                        input_shape=(width, height, depth)))
# kernel_size nur eine Zahl -> Quadratisch
# bei strides analog
# Bei erster Schicht immer noch input_shape mit angeben
# passing = "same" erhält die Größe der Matrix:
#   Vorteil: Details am Rand werden auch beachtet,
#   Nachteil: Mehr Rechnleistung notwendig

model.add(layers.Activation("relu"))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
# model.add(layers.Dropout(0.25))

model.add(layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                        input_shape=(width, height, depth)))
model.add(layers.Activation("relu"))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
# model.add(layers.Dropout(0.25))


model.add(layers.Flatten())  # aufdröseln in einen Spaltenvektor
model.add(layers.Dense(128))  # "Normale" Layer
model.add(layers.Activation("relu"))
model.add(layers.Dense(num_classes))
model.add(layers.Activation("softmax"))  # für WSK


model.summary()

model.compile(
    loss="categorical_crossentropy",
    optimizer=optimizer,
    metrics=["accuracy"])
```

Der model_name wird zu Beginn angegeben, er ist nachher für das Abspeichern und die Grafiken der Lernkurven notwendig, um diese den Modellen zuordnen zu können.

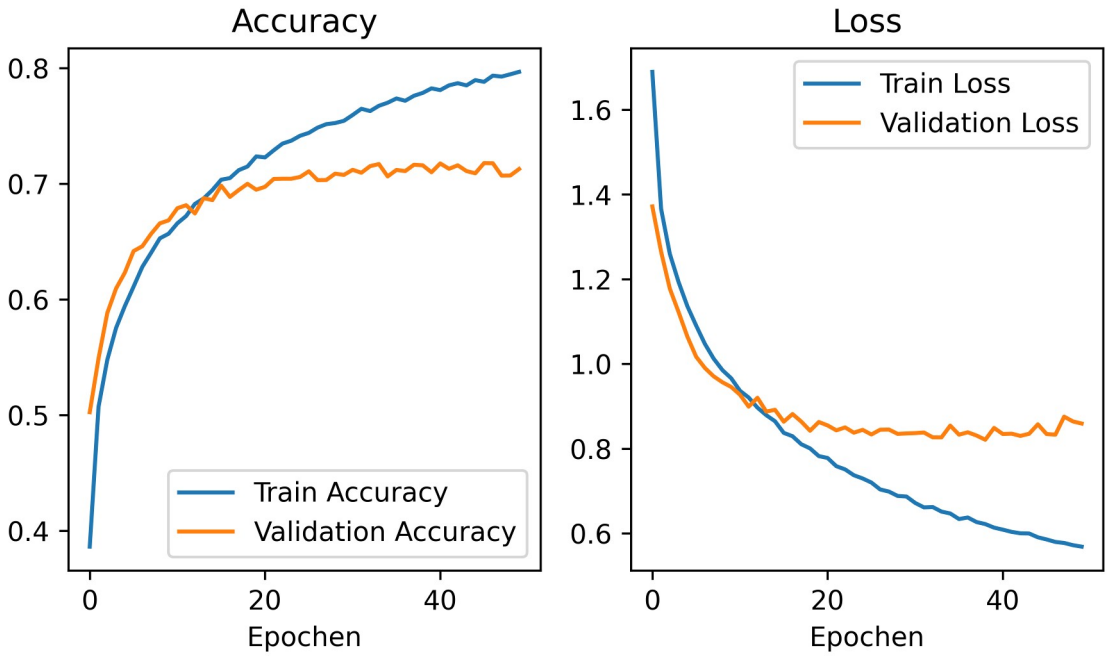
Das Netz selbst besteht aus drei Gruppen. Zwei davon bestehen jeweils aus einer Convolutional Layer und einer MaxPooling Layer. Die Dropout Layer kam erst später dazu. Die letzte Gruppe beinhaltet zwei Dense Layers, deren letzte Schicht die Ausgabe (10 Kategorien) darstellt.
In der allerersten Layer muss noch die Form des Eingangs (input_shape) angegeben werden.
Bevor die Informationen in die Dense Layers gelangen können, muss die zweidimensionale Form, die für die vorigen Schichten notwendig war, in eine eindimensionale umgewandelt werden. Dies macht der Flatten() Befehl.

Ein erster Durchlauf (50 Epochen, Batch Size 100, Learning Rate 0.001) ergab folgendes:



Kurze prinzipielle Erklärung der Plots:
Die „Train Accuracy“ stellt die Genauigkeit der Vorhersage innerhalb der Trainingsdaten dar. Dazu wird der Trainingsdatensatz in zwei Teile aufgeteilt[18] und das Training des ersten Teils wird dann mit dem zweiten getestet. Da sich diese Daten bei jedem Durchgang ändern, trainiert das Netz jedes Mal mit leicht anderen Daten. Dennoch kann es dabei zu einer Überanpassung an die Trainingsdaten kommen, da das System sich ja nur mit diesen kontrolliert. Die „Validation Accuracy“ zeigt, wie genau das trainierte System die nicht in das Training einfließenden Testdaten erkennt. Auf der rechten Seite sieht man den „Loss“, also den Verlust, das ist der Unterschied zwischen den Predicts und den Targets (innerhalb des Trainingsdatensatzes aufgrund der Aufsplittung bzw. im Vergleich mit dem Testdatensatz).
Die Train Accuracy liegt am Ende bei 0,97, die Test Accuracy bei 0,64.
Zum einen sieht man, dass die Trainingskurven gegen Ende schon relativ flach verlaufen und sich einem Sättigungswert annähern. Dies weißt darauf hin, dass 50 Epochen hier ausreichen. Andererseits sieht man klar, dass die Kurven je Diagramm immer weiter auseinander gehen, was auf ein Overfitting, also eine zu starke Anpassung des Netzes an seine Trainingsdaten, hindeuten könnte. Ein Overfitting kann auch darauf hinweisen, dass das Netz für den Zweck zu komplex ist. Dies sollte man zumindest im Hinterkopf behalten.
Deswegen werden im zweiten Versuch die Dropout Schichten mit einer Rate von 25 % eingebaut, da diese einem Overfitting entgegenwirken können.

18 Im Standardfall 80% zu 20 %



Man erkennt, dass die Kurven näher beieinander liegen, das Overfitting wurde durch das Einbauen der Dropout Schichten vermindert.
Die Train Accuracy liegt am Ende bei 0,79, die Test Accuracy bei 0,71. Letzteres ist schon ein ziemlich guter Wert.
Ein Versuch, den Dropout in beiden Schichten auf 0,30 zu erhöhen brachte keine Verbesserung der Ergebnisse.

Eine interessante Möglichkeit der Analyse bei solchen Kategorisierungen ist eine Confusion Matrix. Sie zeigt an, wie viele Bilder aus welcher Kategorie wie einsortiert wurden.
Der Code:

```
# %% Confusion Matrix

predictions = model.predict(test_X)
predicted_labels = np.argmax(predictions, axis=1)
true_labels = np.argmax(test_Y, axis=1)

# Erstellen der Confusion Matrix
cm = confusion_matrix(true_labels, predicted_labels)

print(cm)
```

Die vorhergesagten und die eigentlichen Werte werden in je eine Variable geschrieben und dann in der Confusion Matrix dargestellt.

Für mein erstes Netz sieht die Matrix folgendermaßen aus:

```
313/313 [==============================] - 1s 2ms/step
[[765  22  44  20  21   4  13   8  64  39]
 [ 16 837   3  10   5   3   9   2  23  92]
 [ 69   5 573  67 111  55  72  31  12   5]
 [ 24  13  74 545  77 140  75  28   8  16]
 [ 21   1  69  57 700  35  45  55  11   6]
 [ 13   5  53 221  48 569  33  43   7   8]
 [  8   5  53  50  40  17 813   3   7   4]
 [ 16   3  35  38  52  68   7 767   4  10]
 [ 57  39  11  16  10   6   9   4 818  30]
 [ 20  73   9  21   3   6   9  18  25 816]]
```

Man sieht, die Kategorien 2 und 3 haben Probleme, da bei ihnen auf der Hauptdiagonale (korrekte Einordnung) die kleinsten Werte stehen. 2 (Vogel) wird viel mit 4 (Reh) verwechselt und 3 (Katze) mit 5 (Hund). Ein Grund dafür könnte sein, dass die Bilder von Vögeln und von Rehen oftmals einen „Wald“-Hintergrund haben und sich Hunde und Katzen – zumindest bei dieser Auflösung – schon sehr ähnlich sehen.

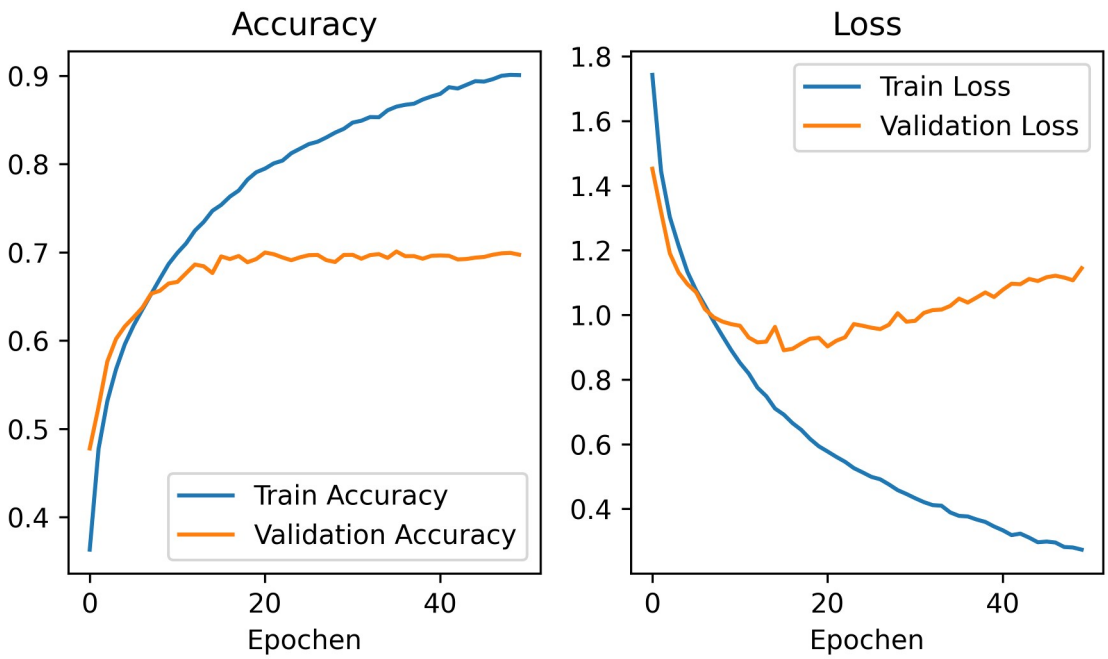
2.3.2.5 Das zweite Netz

Am Ende der zweiten Schichtgruppe laufen 2048 Knoten auf eine Dense Layer mit 128 Knoten auf, die dann in eine Dense Layer mit 10 Knoten mündet. Da es erfahrungsgemäß von Vorteil ist, wenn man die Daten am Ende „trichterförmig“ auf die Ausgabe vorbereitet, setze ich eine zusätzliche Schicht mit 512 Nodes ein.

```
# %% Model 2

model_name = "M2"

model = models.Sequential()

model.add(layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                        input_shape=(width, height, depth)))

model.add(layers.Activation("relu"))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
model.add(layers.Dropout(0.25))

model.add(layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                        input_shape=(width, height, depth)))
model.add(layers.Activation("relu"))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
model.add(layers.Dropout(0.25))


model.add(layers.Flatten())  # aufdröseln in einen Spaltenvektor
model.add(layers.Dense(512))  # "Normale" Layer
model.add(layers.Activation("relu"))
# model.add(layers.Dropout(0.25))
model.add(layers.Dense(128))  # "Normale" Layer
model.add(layers.Activation("relu"))
model.add(layers.Dense(num_classes))
model.add(layers.Activation("softmax"))  # für WSK


model.summary()

model.compile(
    loss="categorical_crossentropy",
    optimizer=optimizer,
    metrics=["accuracy"])
```

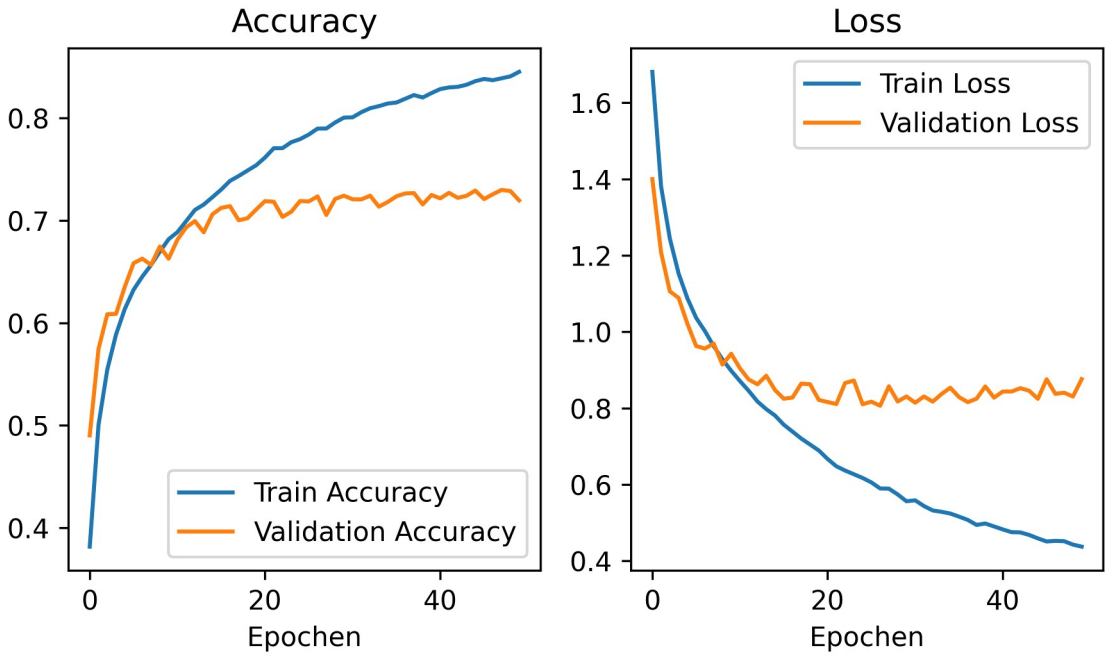
Die Dropout-Schicht wird erst bei einem späteren Versuch aktiv.

Der erste Durchlauf ergibt eine Train Accuracy von 0,90 und eine Test Accuracy von 0,69



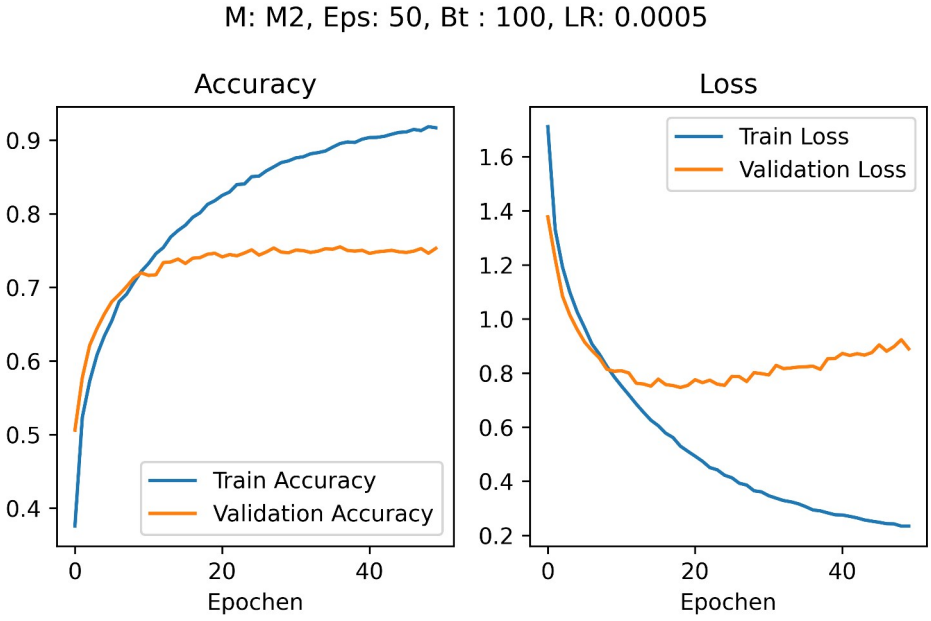
Dies bedeutet eigentlich eine Verschlechterung zum vorigen Netz, weswegen ich nun die Dropout Schicht zwischen den Dense Layers aktiviere, um zu sehen, ob das einen positiven Einfluss hat.

Im zweiten Durchlauf ergibt sich folgendes Bild:



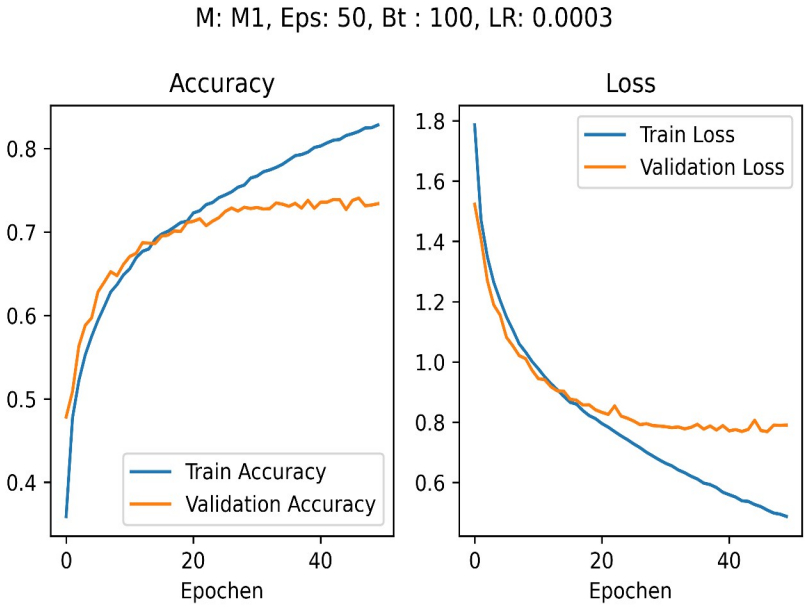
Die Train Accuracy liegt bei 0,84, die Test Accuracy bei 0,72. Das bedeutet, dass die Testgenauigkeit leicht gesteigert werden konnte und die Kurven insgesamt näher beieinander liegen.

Ein weiterer Parameter, der verändert werden könnte, ist die Lernrate:
Eine Erhöhung auf 0,002 erbrachte eine leichte Verbesserung, die Test Accuracy lag hier bei 0,73. Eine Verringerung auf 0,0005 hingegen erhöhte die Test Accuracy auf 0,75, die Train Accuracy stieg auf 0,91.



Ein Wert von 0,0007 brachte keine Verbesserung.
Eine weitere Absenkung auf 0,0003 steigerte noch einmal geringfügig auf knapp unter 0,76. Eine weitere Verringerung auf 0,0002 lies die Genauigkeit auf 0,74 fallen. Ich blieb also bei 0,0003

Ich testete mit diesem Wert noch einmal mein **erstes Netz**. Mit einem Wert von 0,73 bei der Testgenauigkeit entspricht das für sich gesehen einer ziemlichen Verbesserung, kam aber nicht an die Leistung von Netz 2 heran.



Dies deutet darauf hin, dass das zweite Netz wohl doch etwas besser ist, auch wenn die Kurven immer noch auf ein Overfitting oder ein zu komplexes Netz hinweisen.

Die Confusion Matrix ergibt hier folgendes:

```
313/313 [==============================] - 1s 2ms/step
[[801  22  29  18  17   9   8  10  60  26]
 [  7 884   3   7   3   5   4   1  23  63]
 [ 50   4 648  49  84  60  62  19  15   9]
 [ 16   6  58 557  60 183  59  34  15  12]
 [ 12   4  74  46 735  31  41  47   7   3]
 [ 14   4  26 153  40 690  18  34  12   9]
 [  4   7  41  50  29  25 831   2   6   5]
 [ 17   5  23  43  40  60   2 799   3   8]
 [ 45  38  10  11   6   6   8   2 854  20]
 [ 34  75   6  12   6  10   8  13  27 809]]
```

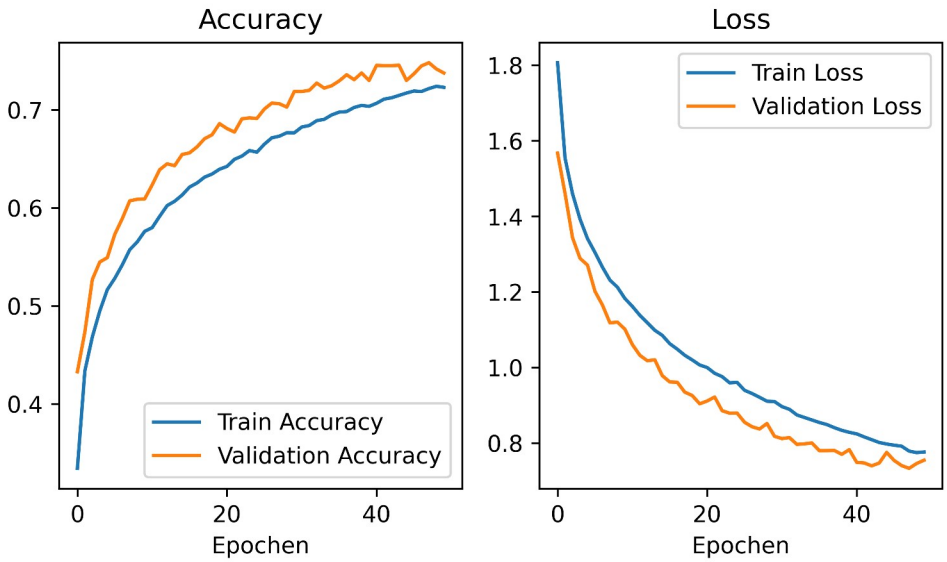
Die Kategorien 2 und 3 sind immer noch die schlechtesten, aber die Anzahl der falschen Einordnungen ist gesunken (bei einer höheren Erkennungsrate verständlich).

2.3.2.6 Das dritte Netz

Aufgrund dieser Erfahrungen versuchte ich ein drittes, noch etwas komplexeres Netz, indem ich noch eine Gruppe mit einer Convolutional, einer MaxPooling und einer Dropout Schicht einbaute. Da aber nach der dritten Gruppe nur noch 512 Ausgangssignale übrig sind, passte ich meine erste darauffolgende Dense Layer auf 256 Nodes an.

```
# %% Model 3

model_name = "03"


model = models.Sequential()

model.add(layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                        input_shape=(width, height, depth)))
model.add(layers.Activation("relu"))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
model.add(layers.Dropout(0.25))

model.add(layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                        input_shape=(width, height, depth)))
model.add(layers.Activation("relu"))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
model.add(layers.Dropout(0.25))

model.add(layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                        input_shape=(width, height, depth)))
model.add(layers.Activation("relu"))
model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
model.add(layers.Dropout(0.25))

model.add(layers.Flatten())  # aufdröseln in einen Spaltenvektor
model.add(layers.Dense(256))  # "Normale" Layer
model.add(layers.Activation("relu"))
model.add(layers.Dropout(0.25))
model.add(layers.Dense(128))  # "Normale" Layer
model.add(layers.Activation("relu"))
model.add(layers.Dense(num_classes))
model.add(layers.Activation("softmax"))  # für WSK


model.summary()

model.compile(
    loss="categorical_crossentropy",
    optimizer=optimizer,
    metrics=["accuracy"])
```

Ein erster Durchlauf mit einer Lernrate von 0,0003 ergab ein interessantes Bild:



Die Trainingsgenauigkeit (am Ende 0,72) treibt die Testgenauigkeit (am Ende 0,73) über den ganzen Verlauf vor sich her! Die Kurven liegen sehr gut beieinander.
Ich versuchte nun als erstes die Lernrate wieder zu erhöhen, um die Auswirkungen zu sehen.
0,0007 erbrachte eine Testgenauigkeit von 0,77, eine weitere Steigerung auf 0,001 (0,76) war keine Verbesserung, ebenso ein Versuch mit 0,0005 (0,77). 0,0006 ergab interessanterweise nur eine Genauigkeit von knapp 0,75. Ich setzte daraufhin die Lernrate wieder auf 0,0007.

Hier noch kurz die Confusion Matrix des dritten Netzes:

```
313/313 [==============================] - 1s 2ms/step
[[799  18  24  11  20   2   8  11  79  28]
 [ 11 899   2   6   2   2   4   0  26  48]
 [ 67   8 591  40 114  54  86  20  12   8]
 [ 16   8  42 533  97 162  81  33  16  12]
 [ 12   3  27  32 820  20  47  33   4   2]
 [  8   2  31 160  61 659  30  38   6   5]
 [  5   3  17  31  38  16 876   4   7   3]
 [ 13   4  23  26  59  35   9 816   3  12]
 [ 41  13   9  13   4   4   2   6 893  15]
 [ 26  73   4  11   4   4   6   7  27 838]]
```

Interessant ist, dass das dritte Netz trotz seiner insgesamt besseren Genauigkeit mit den Kategorien 2 und 3 mehr Probleme hat als Netz 2 und und in Kategorie 3 sogar mehr als Netz 1. Netz 3 ist hauptsächlich in den „hinteren“ Kategorien besser.

2.3.2.7 Hypertuning – der Aufbau

In Keras gibt es eine Funktion, mit der man die Parameter eines neuronalen Netzes automatisch mit verschiedenen Werten testen kann. Ich wollte es mit den Netzen 2 und 3 ver-

suchen. Da dies aber sehr zeitaufwendig ist[19], habe ich die anzupassenden Parameter sehr eingeschränkt.

Für das zweite Netz sieht der Code folgendermaßen aus:

```
# %% Hypertuning Model 2

# Definition der Hyperparameter-Bereiche
hp = HyperParameters()
# hp.Choice('pool_size', values=[(2, 2), (3, 3)])
# hp.Choice('optimizer', values=['adam', 'sgd'])
hp.Float('dropout', min_value=0.2, max_value=0.4, step=0.05)
hp.Int('filters1', min_value=26, max_value=40, step = 2)
hp.Int('filters2', min_value=26, max_value=40, step = 2)


def build_model(hp):
    model = models.Sequential()

    model.add(layers.Conv2D(filters=hp.get('filters1'), kernel_size=3, strides=1, padding="same",
                            input_shape=(width, height, depth)))

    model.add(layers.Activation("relu"))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
    model.add(layers.Dropout(hp.get("dropout")))

    model.add(layers.Conv2D(filters=hp.get('filters2'), kernel_size=3, strides=1, padding="same",
                            input_shape=(width, height, depth)))
    model.add(layers.Activation("relu"))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
    model.add(layers.Dropout(hp.get("dropout")))


    model.add(layers.Flatten())  # aufdröseln in einen Spaltenvektor
    model.add(layers.Dense(512))  # "Normale" Layer
    model.add(layers.Activation("relu"))
    model.add(layers.Dropout(hp.get("dropout")))
    model.add(layers.Dense(128))  # "Normale" Layer
    model.add(layers.Activation("relu"))
    model.add(layers.Dense(num_classes))
    model.add(layers.Activation("softmax"))  # für WSK


    # model.summary()

    model.compile(
        loss="categorical_crossentropy",
        optimizer=optimizers.Adam(learning_rate = 0.0003),
        metrics=["accuracy"])

    return model
```

Zunächst werden die Parameter, die man variabel gestalten will, mit ihren Bereichen und Schrittweiten angegeben. Dann kommt das Netz, in das dann statt der festen Werte die obigen Bereiche geschrieben werden.
Den Optimizer habe ich herausgenommen, da SGD (Stochastic Gradient Descent) bei anderen Probedurchläufen bei dieser Art von Problem durchgehend schlechtere Ergebnisse erbrachte als Adam.
Für die zwei Filter habe ich zwei verschiedene Parameter gewählt. Man könnte hier auch nur einen Filter einzeln nehmen, was aber in einem Testlauf keine Verbesserung meiner Ergebnisse erbrachte.

19 Der Tuner funktioniert in meinem System leider nur auf der CPU und nicht auf der GPU, was ein Leistungsunterschied im Bereich Faktor 10 ist, damit sind Zeiten über 24 Stunden nicht unüblich.

Den Befehl model.summary() habe ich auskommentiert, da ansonsten nach jedem Durchlauf die Daten des Netzes in der Konsole angezeigt werden.

Nun erfolgt die Initialisierung des Tuners:

```
# Erstellung des Tuners und Durchführung der Suche
tuner = Hyperband(
    build_model,
    objective='val_accuracy',
    max_epochs=100,
    hyperparameters=hp,
    directory='tuning',
    project_name='M2')

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=5)
tuner.search(x=train_X, y=train_Y, validation_data=(test_X, test_Y), epochs=10, callbacks=[stop_early])

# Abrufen der besten Hyperparameter-Kombination
best_hp = tuner.get_best_hyperparameters()[0]


print("Beste Hyperparameter:")
for param, value in best_hp.values.items():
    print(f"{param}: {value}")

# Erstellen des finalen Modells mit den besten Hyperparametern
final_model = tuner.hypermodel.build(best_hp)
```

Zu Beginn wird das oben definierte Model eingebunden, dann angegeben, welches der für das Training ausschlaggebende Wert sein soll. Danach erfolgt die Angabe der maximal zu durchlaufenden Epochen pro Test und die Parameter von oben. Zuletzt wird noch ein Ordner und ein Projektname vergeben, unter dem die Daten abgespeichert werden. Dies hat den Vorteil, dass man den Tuner abbrechen und dann an dieser Stelle[20] wieder weiterlaufen lassen kann.

Der Code für das dritte Netz ist analog zum zweiten, der einzige Unterschied ist ein zusätzlicher Parameter für den dritten Filter:

20 Er beginnt nach der letzten vollständig durchlaufenen Konfiguration.

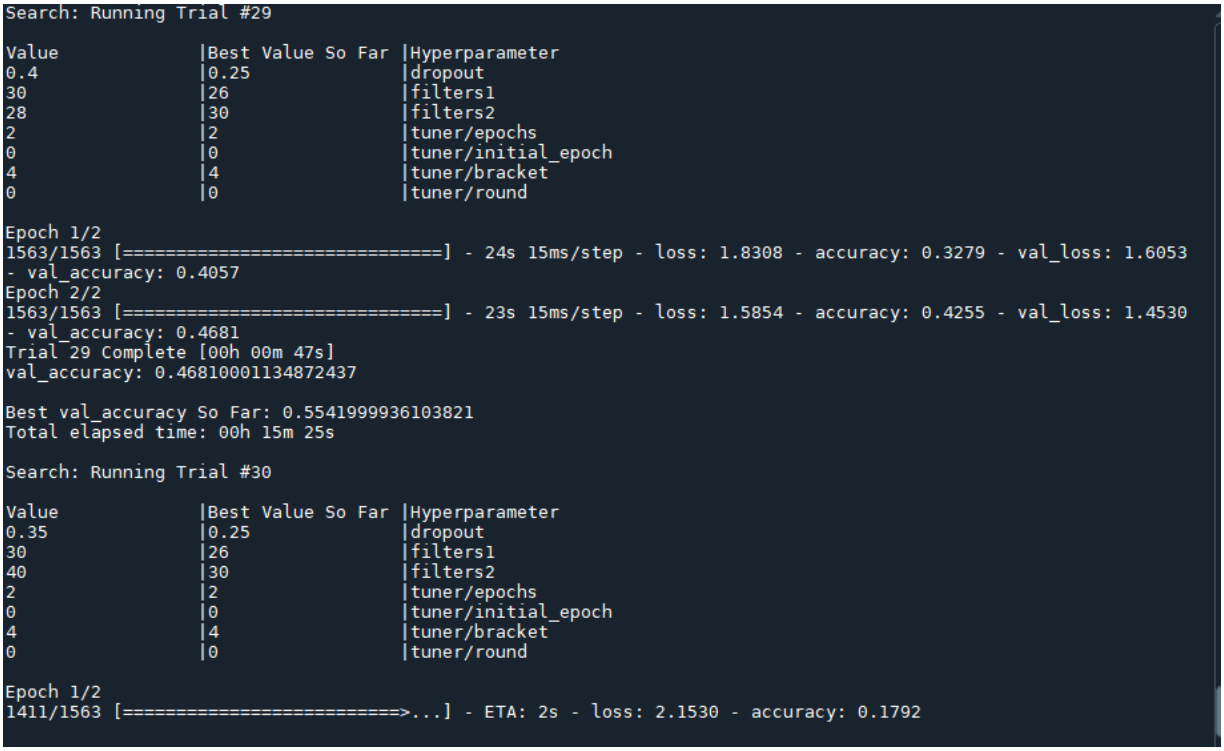
```python
# %% Hypertuning mit Model 3


# Definition der Hyperparameter-Bereiche
hp = HyperParameters()
hp.Float('dropout', min_value=0.2, max_value=0.4, step=0.05)
hp.Int('filters1', min_value=26, max_value=40, step = 2)
hp.Int('filters2', min_value=26, max_value=40, step = 2)
hp.Int('filters3', min_value=26, max_value=40, step = 2)

def build_model(hp):
    model = models.Sequential()

    model.add(layers.Conv2D(filters=hp.get('filters1'), kernel_size=3, strides=1, padding="same",
                            input_shape=(width, height, depth)))
    model.add(layers.Activation("relu"))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
    model.add(layers.Dropout(hp.get('dropout')))

    model.add(layers.Conv2D(filters=hp.get('filters2'), kernel_size=3, strides=1, padding="same",
                            input_shape=(width, height, depth)))
    model.add(layers.Activation("relu"))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
    model.add(layers.Dropout(hp.get('dropout')))

    model.add(layers.Conv2D(filters=hp.get('filters3'), kernel_size=3, strides=1, padding="same",
                            input_shape=(width, height, depth)))
    model.add(layers.Activation("relu"))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding="valid"))
    model.add(layers.Dropout(hp.get('dropout')))

    model.add(layers.Flatten())  # aufdröseln in einen Spaltenvektor
    model.add(layers.Dense(256))  # "Normale" Layer
    model.add(layers.Activation("relu"))
    model.add(layers.Dropout(hp.get('dropout')))
    model.add(layers.Dense(128))  # "Normale" Layer
    model.add(layers.Activation("relu"))
    model.add(layers.Dense(num_classes))
    model.add(layers.Activation("softmax"))  # für WSK

    model.compile(
        loss="categorical_crossentropy",
        optimizer=optimizers.Adam(learning_rate = 0.0007),
        metrics=["accuracy"])

    return model
```

Der Tuner ist ebenfalls – bis auf den Namen des Projekts – gleich:

```python
# Erstellung des Tuners und Durchführung der Suche
tuner = Hyperband(
    build_model,
    objective='val_accuracy',
    max_epochs=100,
    hyperparameters=hp,
    directory='tuning',
    project_name='M3')

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=5)
tuner.search(x=train_X, y=train_Y, validation_data=(test_X, test_Y), epochs=10, callbacks=[stop_early])

# Abrufen der besten Hyperparameter-Kombination
best_hp = tuner.get_best_hyperparameters()[0]


print("Beste Hyperparameter:")
for param, value in best_hp.values.items():
    print(f"{param}: {value}")

# Erstellen des finalen Modells mit den besten Hyperparametern
final_model = tuner.hypermodel.build(best_hp)
```

Ein solcher Tuning-Vorgang hat folgenden Ablauf:
Der Tuner wählt eher zufällig Werte für die einzelnen Parameter aus und testet damit das Netz auf den objective Wert. Zunächst läuft er viele verschiedene Tupel von Parametern mit einer geringen Epochenzahl (hier 2) durch und merkt sich die erreichten Werte. Die besten aus diesem Durchlauf[21] werden dann mit einer höheren Epochenzahl noch einmal gegeneinander getestet. Dabei extrapoliert das System anscheinend auch auf „nahegelegene“ Parameterwerte und testet diese später auch aus. Der Vorgang wiederholt sich über mehrere Runden bis dann ein bestes Model bei voller Epochenzahl gefunden wurde.
Bei höherer Epochenzahl kann die stop_early Bedingung greifen. Sollte sich über die in patience angegebene Epochenzahl die beobachtete Größe nicht positiv ändern, wird der Durchlauf beendet. Dies spart beim Testen Zeit[22].

Die folgenden Bilder zeigt einen Hypertuning-Vorgang[23]: Auf dem ersten Bild sieht man den Ablauf ziemlich zu Beginn, es werden 2 Epochen durchlaufen, die Statistik zeigt jeweils den aktuellen Durchlauf im Vergleich zum bisher besten. Man sieht auch, dass die Batch-Size auf 32 Items festgelegt wurde (1563 Durchgänge pro Epoche).



Das zweite Bild zeigt den Wechsel von 4 Epochen auf 12. Der Tuner ist mit dem Durchlauf von 4 Epochen fertig, die bisher beste Kombination von Parametern trägt die Nummer 0000. Die nächste Runde beginnt bei Epoche 5 von 12, die ersten 4 Epochen wurden ja schon im vorigen Durchgang berechnet.

21 Wie viele es jeweils sind, konnte ich leider nicht herausfinden.
22 Die hier verwendeten Informationen beruhen auf meiner Beobachtung. Leider habe ich im Internet keine genaueren Informationen gefunden.
23 Aus Zeitgründen habe ich hier Bilder von einem früheren Testlauf verwendet.

```
Value             |Best Value So Far |Hyperparameter
0.4               |0.25              |dropout
26                |26                |filters1
26                |30                |filters2
4                 |4                 |tuner/epochs
2                 |2                 |tuner/initial_epoch
4                 |4                 |tuner/bracket
1                 |1                 |tuner/round
0091              |0000              |tuner/trial_id

Epoch 3/4
1563/1563 [==============================] - 22s 14ms/step - loss: 1.5195 - accuracy: 0.4517 - val_loss: 1.4102
- val_accuracy: 0.4874
Epoch 4/4
1563/1563 [==============================] - 21s 13ms/step - loss: 1.4690 - accuracy: 0.4738 - val_loss: 1.3773
- val_accuracy: 0.5051
Trial 131 Complete [00h 00m 44s]
val_accuracy: 0.5051000118255615

Best val_accuracy So Far: 0.5774999856948853
Total elapsed time: 01h 49m 58s

Search: Running Trial #132

Value             |Best Value So Far |Hyperparameter
0.25              |0.25              |dropout
26                |26                |filters1
30                |30                |filters2
12                |4                 |tuner/epochs
4                 |2                 |tuner/initial_epoch
4                 |4                 |tuner/bracket
2                 |1                 |tuner/round
0098              |0000              |tuner/trial_id

Epoch 5/12
1563/1563 [==============================] - 24s 15ms/step - loss: 1.1458 - accuracy: 0.5931 - val_loss: 1.0866
- val_accuracy: 0.6153
Epoch 6/12
 970/1563 [=================>............] - ETA: 7s - loss: 1.1022 - accuracy: 0.6086
```

Man kann auch erkennen, dass der Tuner für die Erhöhung der Epochenzahl als erstes im Prinzip das Parametertupel gegen sich selbst antreten lässt – nur eben mit höherer Epochenzahl.

Und hier noch ein Bild, auf dem man sieht, dass das stop_early greift:

```
Epoch 22/34
1563/1563 [==============================] - 26s 16ms/step - loss: 0.7723 - accuracy: 0.7255 - val_loss: 1.0255
- val_accuracy: 0.6515
Epoch 23/34
1563/1563 [==============================] - 23s 15ms/step - loss: 0.7578 - accuracy: 0.7318 - val_loss: 1.0274
- val_accuracy: 0.6603
Epoch 24/34
1563/1563 [==============================] - 22s 14ms/step - loss: 0.7478 - accuracy: 0.7332 - val_loss: 1.0117
- val_accuracy: 0.6540
Epoch 25/34
1563/1563 [==============================] - 22s 14ms/step - loss: 0.7414 - accuracy: 0.7367 - val_loss: 0.9899
- val_accuracy: 0.6653
Epoch 26/34
1563/1563 [==============================] - 22s 14ms/step - loss: 0.7245 - accuracy: 0.7411 - val_loss: 1.0544
- val_accuracy: 0.6566
Epoch 27/34
1563/1563 [==============================] - 24s 15ms/step - loss: 0.7174 - accuracy: 0.7436 - val_loss: 1.0227
- val_accuracy: 0.6583
Epoch 28/34
1563/1563 [==============================] - 21s 14ms/step - loss: 0.7144 - accuracy: 0.7468 - val_loss: 1.0213
- val_accuracy: 0.6591
Epoch 29/34
1563/1563 [==============================] - 21s 14ms/step - loss: 0.6984 - accuracy: 0.7514 - val_loss: 1.0468
- val_accuracy: 0.6547
Epoch 30/34
1563/1563 [==============================] - 21s 14ms/step - loss: 0.6918 - accuracy: 0.7530 - val_loss: 1.0645
- val_accuracy: 0.6492
Trial 143 Complete [00h 07m 15s]
val_accuracy: 0.6653000116348267

Best val_accuracy So Far: 0.6653000116348267
Total elapsed time: 02h 35m 25s
```

2.3.2.8 Hypertuning – Ergebnisse

Das Hypertuning war bei beiden Netzen sehr zeitaufwendig, da es, wie bereits in einer Fußnote erwähnt, nur auf meinen Ryzen 5 3600X zugreift[24] und nicht auf die Nvidia GTX 1080.

Das Training des zweiten Netzes habe ich nach 9 Stunden bei folgenden Werten abgebrochen[25]:

```
Best val_accuracy So Far: 0.76419997215271
Total elapsed time: 09h 05m 33s

Search: Running Trial #219

Value             |Best Value So Far |Hyperparameter
0.4               |0.25              |dropout
26                |36                |filters1
34                |38                |filters2
12                |34                |tuner/epochs
0                 |12                |tuner/initial_epoch
2                 |4                 |tuner/bracket
0                 |3                 |tuner/round
```

Ein Durchlauf mit diesen Werten erbrachte bei 50 Epochen leider nur eine leichte Verbesserung gegenüber meinen von Hand gefundenen Werten. Die Testgenauigkeit lag knapp über 0,76.

Nach über 10 Stunden und 30 Minuten wurde die Optimierung von Netz 3 beendet. Die demnach besten Parameter waren:

```
Best val_accuracy So Far: 0.7829999923706055
Total elapsed time: 00h 25m 08s
INFO:tensorflow:Oracle triggered exit
Beste Hyperparameter:
dropout: 0.2
filters1: 32
filters2: 40
filters3: 38
tuner/epochs: 100
tuner/initial_epoch: 34
tuner/bracket: 4
tuner/round: 4
tuner/trial_id: 0143
```

Ein Training mit diesen Daten ergab dass das Netz die Testgenauigkeit von 0,78 bei 50 Epochen zwar dreimal übertraf, am Ende aber auf 0,7746 landete. Auch ein paar veränderte Batch Sizes (32 und 65 statt 100) brachten keine Verbesserung.

Entweder habe ich von Hand bereits das Beste aus den Netzen herausgeholt, oder man müsste mehr Parameter in einem jeweils weiteren Spektrum testen. Da dies aber mit meinen beschränkten zeitlichen und hardwaretechnischen Mitteln nicht möglich ist, bleibt es bei einer maximalen Testgenauigkeit von etwas über 0,77.

---

24 Dies ist leider für Versionen seit 2.11 nicht mehr der Fall, bei diesen gibt es für Windows keine Unterstützung mehr, dies ist aber kein gewolltes „feature“, sondern die Requests wurden bisher nicht erfüllt.

25 Die Datenmenge des Tuning-Ordners betrug zu diesem Zeitpunkt 2,91 GB!

2.3.3 Versuch der Kategorisierung von selbst erstellten Bildern

Um die Modelle auch mal außerhalb der Laborbedingungen des CIFAR-10-Datensatzes zu testen, habe ich mir ein paar Bilder zurechtgeschnitten (quadratisches Format) und entsprechend verkleinert (32 mal 32 Pixel). Diese sollten dann eingelesen, in die für die Netze notwendige Form gebracht und getestet werden.

2.3.3.1 Die Bilder

Ich habe mich für folgende Bilder entschieden (die kleinen Versionen wurden absichtlich etwas größer dargestellt):

Porsche:

Bei diesem Bild ist das Auto zwar zu erkennen, der Kontrast zum Hintergrund ist aber nicht besonders hoch, auch das Haus hat Strukturen.

Playmobil:

Hier auch ein Porsche – zwar ein Spielzeug, aber eine gute Nachbildung – und mit sehr gutem Kontrast zum Hintergrund.

Hund1:  

Ein Hund, trotz des eher geringen Kontrasts gut zu erkennen.

Hund2:  

Das könnte schwer werden – weißer Hund auf weißem Hintergrund. Die Schnauze, die Augen und die Ohren sind jedoch charakteristisch.

Hirsch:  

Ein „typscher“ Hirsch, wie er sicher auch im Datensatz vorkommt. Er sollte keine Probleme bereiten.

Lotti:  

Lotti ist eines unserer Zwergkaninchen, auf die Einsortierung bin ich gespannt – auch wegen des geringen Kontrastes im Vergleich zum Hintergrund.

Der Code für das Vorhaben sieht folgendermaßen aus:

```
# %% Einlesen eines jpg-Bildes und Umwandeln in ein Numpy-Array

images = ["Porsche-klein.png", "Playmobil-klein.png","Hund1-klein.jpg",
          "Hund2-klein.jpg","Hirsch-klein.jpg", "Lotti-klein.jpg"]


image_path = "Bilder"

categories = ['Flugzeug', 'Auto', 'Vogel', 'Katze', 'Hirsch', 'Hund', 'Frosch', 'Pferd', 'Schiff', 'LKW']

def show_image(image_path):
    img = mpimg.imread(image_path)
    plt.imshow(img)
    plt.axis('off')
    plt.show()


def process_image(image_path):
    image = cv2.imread(image_path) # Bild einlesen
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Bild in das RGB-Farbformat konvertieren
    normalized_image = rgb_image.astype('float32') / 255.0  # Bild in einen Float32-Array umwandeln und dan
    reshaped_image = np.expand_dims(normalized_image, axis=0)  # Bild in die Shape des CIFAR-10-Datensatzes
    return reshaped_image


def predict_image(image_path, model):
    img = process_image(image_path)
    prediction = model.predict(img)
    predicted_class = np.argmax(prediction)
    return predicted_class


for image in images:
    image_con = os.path.join(image_path, image)
    show_image(image_con)
    processed_image = process_image(image_con)
    predicted_class = predict_image(image_con, model)
    predicted_label = categories[predicted_class]
    print(image, ":|t", predicted_label)
```

Zunächst werden die Dateinamen in einer Liste und der Pfad in einer Variablen abgelegt.
Die darauffolgende Funktion zeigt ein Bild testweise an.
In der nächsten Funktion wird das Bild in die für das neuronale Netz notwendige Form eines Arrays (1,32,32,3) gebracht.

Danach erfolgt die Vorhersagefunktion. Der Befehl np.argmax() sucht aus dem Vektor die größte Zahl heraus. Das entspricht der Entscheidung für die wahrscheinlichste Kategorie.
Und zuletzt wird alles zusammen in eine for-Schleife gepackt, damit in der Konsole alle Vorhersagen auf einmal angezeigt werden.

Die Ergebnisse beim Test gegen die einzelnen neuronalen Netze – jeweils mit ihren besten Parametern – sehen wie folgt aus (in Klammern jeweils die Testgenauigkeit, die ich erreicht hatte):

Bild	Model 1 (0.73)	Model 2 (0.76)	Model 3 (0.77)
Porsche	Auto √	Auto √	Schiff
Playmobil	Auto √	Auto √	Auto √
Hund 1	Hund √	Hund √	Hund √
Hund 2	Flugzeug	Flugzeug	Flugzeug
Hirsch	Hirsch √	Hirsch √	Hirsch √
Außer Konkurrenz:			
Lotti	LKW	Hund	Hund

Alle drei Netze sind in ihrer Einschätzung ähnlich. Interessanterweise patzt Netz 3 einmal beim Porsche, Dass Hund 2 nicht als Hund erkannt wird, war abzusehen – ein weißer Hund auf weißem Hintergrund ist schwer. Dennoch hatte ich gehofft, dass die Schnauze und die Augen als typisches Merkmal erkannt werden. Anscheinend haben sich die Netze vom hohen Weißanteil täuschen lassen[26].
Bei Lotti tun sich alle drei schwer, was kein Wunder ist – immerhin ist ein Kaninchen nicht vorgesehen. Interessant finde ich, dass Lotti ein LKW sein soll …

2.4 Ausblick

Die Beschäftigung mit den CNNs hat mir sehr viel Spaß gemacht, ich habe dabei viel an Erfahrung und auch an neuen Aspekten dazugewonnen. Ich habe im Vorfeld noch viele weitere Netzstrukturen an dem CIFAR-10 Datensatz ausprobiert (LeNet[27], MiniVGG)[28] und mich dafür auch im Internet eingelesen, aber eine Behandlung dieser würde den – sowieso schon gut überzogenen – Rahmen dieser Arbeit sprengen.

Abschließend kann ich sagen, dass ich mit meinen Modellen sehr zufrieden bin. Wenn man sich im Internet umschaut, gibt es sicher viele Netze, die auf eine Testgenauigkeit von 90 % kommen, diese sind aber wesentlich komplexer und wurden sicher mit entsprechender Rechenpower über viele Epochen optimiert.
Eine wichtige Sache, die ich aus dieser Arbeit mitnehme ist, dass das Optimieren solcher Netze ganz oft von mindestens einem der folgenden Dinge abhängt:

26 Flugzeuge werden gerne weiß lackiert, weil es sich dann nicht so stark aufheizt und so die notwendige Kühlleistung verringert wird.
27 Dieses habe ich auch von Hand optimiert (Test Accuracy 68,7 %) und habe es dann durch den Tuner laufen lassen. Nach über 9 Stunden kam am Ende auch nicht auf bessere Werte.
28 Im Quellcode habe ich die zusätzlichen Netze gelassen, allerdings werden sie in dieser Arbeit nicht weiter erwähnt.

- Gefühl und Vorahnung, bzw. Erfahrung
- Rechenpower
- Zufall

Dass die Testgenauigkeit dann doch eher gering bleibt (bei der Erkennung von handschriftlichen Ziffern lagen meine Netze bei 99 %), liegt wohl an der wirklich schlechten Qualität der Bilder – auf 32 mal 32 Pixeln in Farbe ist es schwer, einen Hund von einer Katze zu unterscheiden oder einen braunen Hirsch vor braunem Hintergrund überhaupt erst einmal zu erkennen.