Keras - Eine Einführung in das Maschinelle Lernen mit Tensorflow

Joshua Roschlaub - Hamburger Sternenwarte

August 2021

Contents

1 Einleitung

Keras ist eine Open Source Bibliothek, mit welcher der Einstieg in das Arbeiten mit Machine Learning so unkompliziert wie möglich gehalten werden soll. Deswegen ist diese API besonders benutzerfreundlich gestaltet und erlaubt, in nur wenigen Schritten von der Idee zur Implementation und Verwendung eines Neuronalen Netzwerkes zu gelangen. Keras ist in Python geschrieben und seit Tensorflow 1.4 Teil der Tensorflow Core API.

Dieser Leitfaden soll für einen komfortablen Einstieg in die Verwendung von Keras und die Arbeit mit Machine Learning sorgen. Dazu wird in Kapitel 2 zunächst ein Überblick über die konzeptionelle Funktionsweise von Neuronalen Netzen gegeben. In Kapitel 3 wird die Verwendung von Keras anhand eines einfachen Beispiels, welches sich mit der automatischen Erkennung von handschriftlich geschriebenen Ziffern beschäftigt, Schritt für Schritt erläutert. Dieses Netzwerk wird in Kapitel 4 mithilfe einer neuen Art von Ebenen, der convolutional layer, erweitert und in seiner Funktionsweise verbessert. Das Kapitel 5 beschäftigt sich dann mit dem HPC (High Performance Computing) Rechenzentrum der Universität Hamburg, welches häufig auch als Hummel Cluster bezeichnet wird. Dabei wird ein Überblick in die Funktionsweise des Clusters gegeben sowie detailliert erklärt, wie die Arbeit mit Tensorflow auf dem Cluster möglich ist. Eine letzte Anwendung finden diese gewonnenen Kenntnisse zum Schluss in Kapitel 6. Dort werden Spektraldaten der Sloan Digital Sky Survey (SDSS) mithilfe von Machine Learning klassifiziert.

2 Die Funktionsweise von Neuronalen Netzen

3 Das erste eigene Netzwerk

3.1 Installation von Tensorflow

Das Keras Paket ist seit 2017 ein Teil der Kernbibliothek von Tensorflow. Das bedeutet, dass Keras vollständig in der Installation von Tensorflow enthalten ist. Tensorflow kann sowohl mit pip, als auch mit conda über einen Konsolenbefehl installiert werden.

```
pip install tensorflow
```

```
conda install tensorflow
```

3.2 GPU Unterstützung

Dieses Kapitel beschäftigt sich mit der GPU Unterstützung von TensorFlow und der implementierten Keras API und damit, wie Code auf einer GPU ausgeführt werden kann.

Code aus der TensorFlow Bibliothek wird ohne zusätzliche Eingabe auf der GPU laufen. Die einzige Voraussetzung dafür ist eine CUDA-fähige Grafikkarte, also eine NVIDIA-GPU der G8x Generation oder neuer. CUDA-fähig sind außerdem alle Karten der GeForce, Quadro und Tesla Reihen. Ist diese Voraussetzung erfüllt und erkennt TensorFlow sowohl eine CPU als auch eine GPU, wird automatisch die GPU verwendet. Ist dies nicht gewünscht, besteht trotzdem die Möglichkeit den Code auf der CPU ausführen zu lassen.

3.3 Vorbereitung der Trainings- und Test-Daten

3.3.1 Laden der Datensätze

Zu Beginn muss geklärt werden, welchen Typ der Datensatz hat, mit dem gearbeitet werden soll. In diesem Beispiel soll ein Netzwerk erstellt werden, mit dem handschriftlich geschriebene Ziffern erkannt werden sollen. Um das Netzwerk zu trainieren und anschließend testen zu können, wird ein ausreichend großer Datensatz benötigt. Hierzu bietet sich der MNIST Datensatz von handschriftlich geschriebenen Ziffern an, welcher aus einem Trainings- und einem Testset besteht. Das Trainingsset beinhaltet 60.000 Bilder und das Testset weitere 10.000. Alle Bilder sind dabei im selben 28x28 Format. Zu jedem einzelnen Bild gibt es ein zugehöriges label, welches einen ganzzahligen Wert zwischen 0 und 9 hat.

Das Laden der beiden Datensätze ist mit Keras sehr unkompliziert möglich. Zunächst muss tensorflow importiert werden, wobei optional ein alternative Bezeichnung gewählt werden kann.

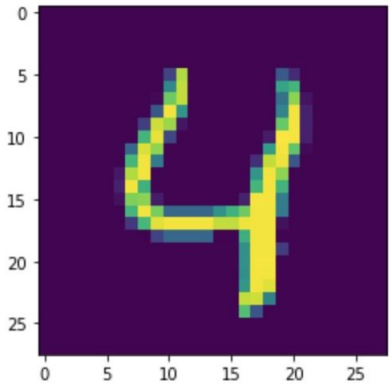
```
import tensorflow as tf
from tensorflow import keras
```

Anschließend kann der MNIST Datensatz geladen werden.

```
mnist = keras.datasets.mnist
(train_samples, train_labels) = mnist.load_data()[0]
(test_samples, test_labels) = mnist.load_data()[1]
```

Die Liste train_samples beinhaltet die Bilddaten der 60.000 Trainingsbilder und hat den shape (60000, 28, 28). Die Liste train_labels beinhaltet die zugehörigen labels und hat somit den shape (60000,). Die beiden Listen test_samples und test_labels beinhalten entsprechend die Bilddaten und labels der 10.000 Bilder aus dem Test-Datensatz. Mit dem Befehl plt.imshow() können die Bilder in der Konsole ausgegeben werden. Der folgende Code stellt die ersten 10 Bilder aus dem Trainingsdatensatz und die zugehörigen labels dar.

```
for i in range(10):
    plt.imshow(test_samples[i])
    plt.show()
    print("label:", test_labels[i])
```




```
label: 4
```

Figure 1: Bild einer handgeschriebenen Ziffer aus dem MNIST Datensatz mit dem zugehörigen label.

3.3.2 Normierung der Daten

Die einzelnen Einträge in den Bildern der hangeschribenen Ziffern aus dem MNIST Datensatz sind ganzzahlige Schwarz-Weiß-Werte zwischen 0 und 255. Möchte man Daten zum trainieren eines Neuronalen Netzwerks verwenden, ist es allgemein sinnvoll diese Daten vorher zu normieren. Damit ist gewährleistet,

dass Daten mit unterschiedlichen Größenordnung besser miteinander kombiniert werden können und insbesondere das Training bestmöglich funktioniert. Um das zu verstehen, betrachten wir ein einfaches Beispiel. Angenommen es soll versucht werden zu ermitteln, ob einem Kunden ein Darlehen gegeben werden soll. Die zur Verfügung stehenden Variablen seien Altern und Einkommen. Sei die Gleichung von folgender Form:

$$Y = Gewicht1 * Alter + Gewicht2 * Einkommen + Konstante$$

Das Alter und Einkommen liegen aber in einer völlig anderen Größenordnung. Somit sind auch die Gewichte 1 und 2 nicht mehr vergleichbar und es nicht eindeutig, ob dem Alter oder dem Einkommen eine größere Bedeutung zugeordnet wird. Um die Gewichte und Daten auf ein vergleichbare Größenordnung zu bringen, werden die Daten vor dem Training normiert. In der Regeln erhöht das sowohl die Treffergenauigkeit des Netzwerks, als auch die Geschwindigkeit des Trainings.
Für die Normierung der Daten kann die normalize() Funktion von Keras genutzt werden. Dadurch werden die Daten auf eine Größe zwischen 0 und 1 skaliert.

```
train_samples = tf.keras.utils.normalize(train_samples, axis=1)
test_samples = tf.keras.utils.normalize(test_samples, axis=1)
```

3.4 Erstellen des Netzwerks

Nun sind die Trainings- und Test-Datensätze in Form von Numpy-arrays bereitgestellt und es kann ein einfaches Netzwerk erstellt werden, welches anschließend mit diesen Daten trainiert und getestet wird. Um die Lesbarkeit des Codes zu erhöhen, bietet es sich an, folgende Module der Keras API einzeln zu importieren.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense, Flatten,
                                    Conv2D, MaxPool2D
from tensorflow.keras.metrics import Accuracy
```

Dadurch kann beispielsweise die Sequential() Funktion ohne vorhergehende Abhängigkeiten aufgerufen werden. In diesem Beispiel wird zunächst ein Neuronales Netzwerk der Klasse Sequential erstellt.

```
model = Sequential([
    Flatten(),
    Dense(units=128, activation='relu'),
    Dense(units=10, activation='softmax')
])
```

Die erste Layer beinhaltet keinerlei Parameter. Sie wandelt die eingegebenen Daten von einem mehrdimensionalen Vektor in einen eindimensionalen Vektor um. Dadurch werden die Bilddaten von der Form (28,28) in die Form (784,)

gebracht.
Die zweite Layer ist eine Dense Layer und beinhaltet insgesamt 128 Neuronen. Diese Ebene beinhaltet somit 100.480 Parameter, da jedes Neuron aus der zweiten Ebene einen einzelnen bias sowie für jedes Neuron aus der ersten Ebene ein weiteres Gewicht benötigt. Als activation function wird die ReLU Funktion verwendet, welche positive Werte unverändert lässt und negative Werte Null setzt.
Auch die dritte Layer ist eine Dense Layer. Hier befinden sich die zehn Ausgabe-Neuronen, wodurch dem Modell weitere 1408 Parameter hinzugefügt werden. Als activation function bietet sich nun die softmax funtion an. Diese hat einen Wertebereich zwischen 0 und 1, wobei sich alle Ausgabewerte dieser Ebene zu 1 addieren.

Bevor das Modell trainiert werden kann, muss die compile Funktion aufgerufen werden. Dabei werden die loss-function, der optimizer sowie die beim Training zu verwendene Metrik festgelegt.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

3.5 Trainieren des Netzwerks

In diesem Abschnitt wird das erstellte Modell mit den Trainings-Daten trainiert. Dafür wird die fit Funktion verwendet. Als Parameter wird der Funktion der Trainings-Datensatz, die Label des Trainings-Datensatzes und die Anzahl der Epochen übergeben. Durch den validation_split wird der Trainings-Datensatz in zwei Teile aufgespalten. Mit den ersten 90% der Daten wird das Modell im Anschluss in 30 Epochen trainiert werden. Mit den letzten 10% der Daten wird das Modell während des Trainings getestet, um auf ein Overfitting zu überprüfen. Dieser Teil der ursprünglichen Trainings-Daten wird somit nicht mehr zum trainieren verwendet. Die batch_size reguliert, mit wie vielen Elementen der Trainings-Daten gleichzeitig trainiert wird. Der Boolean shuffle ist standardmäßig auf True eingestellt. Dies sorgt dafür, dass der Trainings-Datensatz vor jeder Epoche gemischt wird. Dies geschieht nach der Auswahl des validation_split, sodass sich die validation-Daten während des Trainings nicht ändern.

```
history = model.fit(train_samples, train_labels, epochs=30,
            validation_split=0.1, batch_size=120, shuffle=True)
```

In der Variable History werden die Treffergenauigkeiten sowie Verlust-Werte nach den einzelnen Epochen gespeichert. Auf diese Daten kann zugegriffen werden, um eine graphische Darstellung der Treffer-Genauigkeit und Loss-Funktion von Trainings- und Validation-Daten zu erhalten. Diese sind in Abbildungen 2 und 3 dargestellt. Wichtig ist, dass hierfür ein validation-Split der Trainings-Daten notwendig ist.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='upper left')
plt.show()
```

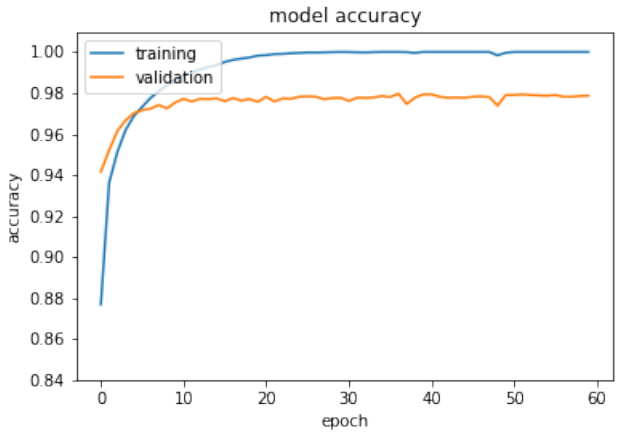


Figure 2: Kurve der Treffergenauigkeit des Modells mit Trainings- und Validation-Daten.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='upper left')
plt.show()
```

In der Abbildung 2 ist zu sehen, dass die Treffergenauigkeit des Modells bei den Trainings-Daten annähernd bei 1 ist. Gleichzeitig liegt die Treffergenauigkeit bei den validation-Daten etwa 2% niedriger. Außerdem ist in Abbildung 3 zu erkennen, wie die loss-Kurve der validation-Daten nach den ersten Epochen kontinuierlich steigt. Beides sind klare Indizien für ein sogenanntes Overfitting, was soviel bedeutet wie, dass das Modell immer schlechter generalisieren kann.

3.6 Testen des Netzwerks

Nun ist das Netzwerk trainiert und das Modell kann benutzt werden, um Vorhersagen über noch unbekannte Daten treffen zu können. Dazu kann die predict-Funktion

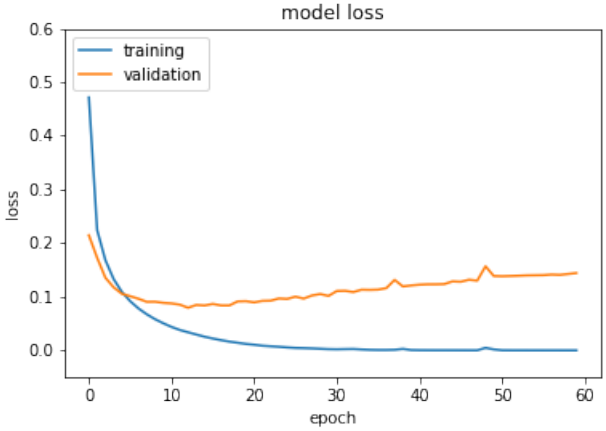


Figure 3: Kurve der Loss-Funktion des Modells mit Trainings- und Validation-Daten.

von Keras aufgerufen werden. Diese nimmt als Eingabe die Liste von Test-Daten und gibt als Ausgabe die Werte der Neuronen aus der output-Layer zurück. In diesem Fall also eine Liste mit zehn Werten zwischen 0 und 1. Die Summe dieser zehn Werte ergibt auch hier wieder 1, da bei der Erstellung des Netzwerks für die activation function der output-Layer softmax gewählt wurde.

```
predictions = model.predict(x=test_samples)
```

Die Liste predictions beinhaltet nun 10.000 Einträge von Listen mit je 10 Werten, also zu jedem Test-Bild die Werte der 10 output-Neuronen. Um daraus die vom Modell getroffenen Vorhersagen abzuleiten, können die Werte der output-Neuronen als Wahrscheinlichkeiten aufgefasst werden. Interessant ist also nur, welcher dieser Werte am größten ist. Mithilfe der Numpy-Funktion argmax kann eine neue Liste der Ziffern mit den jeweils größten Wahrscheinlichkeiten erstellt werden.

```
rounded_predictions = np.argmax(predictions, axis=-1)
```

Da die wahren labels der Test-Daten bekannt sind, lässt sich auch die Treffergenauigkeit dieser Vorhersagen ausgeben. Dazu kann die Klasse `Accuracy` von Keras verwendet werden. Diese ermöglicht einen Vergleich der wahren labels mit den Vorhersagen und eine Ausgabe der Treffergenauigkeit.

```
accuracy = Accuracy()
accuracy.update_state(y_true=test_labels, y_pred=rounded_predictions)
accuracy.result().numpy()
```

3.7 Die Confusion Matrix

Eine Confusion Matrix ist eine anschauliche Art die Funktionsweise eines Netzwerks zu überprüfen. Die Idee ist relativ simpel: Durch gegenüberstellen von Labels und Vorhersagen kann nachvollzogen werden, wie häufig eine Ziffer richtig vorhergesagt und wie häufig sie für eine andere Ziffer gehalten wurde. Mit der Bibliothek Scikit-learn lässt sich eine solche Confusion Matrix unkompliziert ausgeben. Nach der Installation kann das Modul confusion_matrix importiert werden.

```
from sklearn.metrics import confusion_matrix
```

Anschließend kann eine neue Confusion Matrix angelegt werden. Als Parameter müssen die wahren labels und die vorhergesagten Ziffern übergeben werden.

```
cm = confusion_matrix(y_true=test_labels, y_pred=rounded_predictions)
```

Mit dem folgenden Code von der offiziellen Scikit-learn Website kann die Confusion Matrix optisch ansprecheng dargestellt werden. Dort wird die Funktion plot_confusion_matrix definiert, die hier in einer leicht veränderten Version benutzt wird.

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion Matrix',
                          cmap=plt.cm.Blues):
    """
    Diese Funktion printet und plottet die Confusion Matrix
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print("Confusion matrix, without normalization")

    print(cm)

    thresh = cm.max() / 2
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
```

```
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

Beim Aufrufen dieser Funktion müssen zusätzlich zu der Confusion Matrix die labels übergeben werden, in diesem Fall eine Liste mit den Ziffern von 0 bis 9. Außerdem kann mit dem Parameter `normalize` ausgewählt werden, ob die absoluten oder normierten Werte in der Matrix zu sehen sein sollen.

```
cm_plot_labels = range(10)
plot_confusion_matrix(cm=cm, classes=cm_plot_labels,
                      title='Confusion Matrix', normalize=False)
```

Ein mögliches Ergebnis ist in Abbildung 4 dargestellt. Hieraus lassen sich unter Umständen interessante Schlüsse ziehen. Beispielsweise, dass die 5 relativ häufig für eine 3 gehalten wird und dass die 0 im Vergleich zu anderen Ziffern besonders häufig richtig erkannt wird.

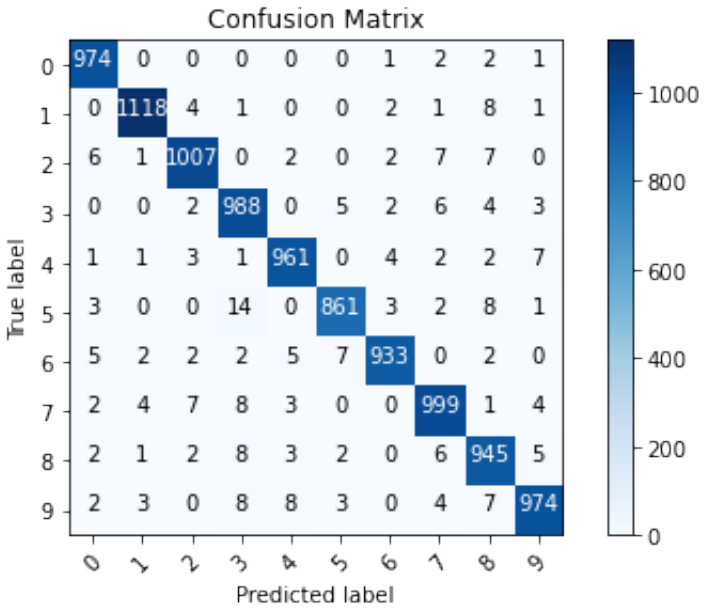


Figure 4: Confusion Matrix des Netzwerks zur Erkennung von handschriftlich geschriebenen Zahlen.

3.8 Netzwerke Speichern und Laden

Es gibt eine Vielzahl an Möglichkeiten ganze Netzwerke, lediglich die Gewichte oder auch nur die Modell-Strukturen abzuspeichern, um sie zu einem späteren

Zeitpunkt laden zu können. Eine kleine Auswahl dieser Möglichkeiten wird im folgenden Kapitel vorgestellt.

3.8.1 model.save

Mit der `model.save` Funktion von Keras kann das gesamte Netzwerk gespeichert werden. Dies beinhaltet die Struktur, die Gewichte, die Trainingskonfigurationen, welche im compiler festgelegt wurden, und den Status des Optimizers, wodurch dass Training auch bei einem Abbruch wieder fortgesetzt werden kann. Die `model.save` Funktion speichert alle diese Daten in einer H5-Datei ab.

```
model.save('digit_recognizer_model.h5')
```

Das Netzwerk kann anschließend mit Hilfe der `load\_model` Funktion unter einem anderen Namen geladen werden.

```
from tensorflow.keras.models import load_model
model1 = load_model('digit_recognizer_model.h5')
```

3.8.2 model.to_json

Mithilfe der `model.to\_json` Funktion wird ausschließlich die Struktur des Modells, also die Anzahl und Konfiguration der Ebenen, in einem json-String abgespeichert.

```
json_string = model.to_json()
```

Mit der `model\_from\_json` Funktion von Keras kann der json-String eingelesen und ein neues Modell mit der alten Struktur, aber neuen Gewichten erstellt werden.

```
from tensorflow.keras.models import model_from_json
model2 = model_from_json(json_string)
```

3.8.3 model.save_weights

Die `model-save\_weights` Funktion von Keras ermöglicht es, die Gewichte eines Netzwerks abzuspeichern. Auch hierfür wird wieder eine H5 Datei verwendet.

```
model.save_weights('my_weights.h5')
```

Zum Einlesen der Gewichte muss zunächst wieder ein neues Modell erstellt werden. Die Struktur dieses neuen Modells muss zur Anzahl der abgespeicherten Gewichte passen.

```
model3 = Sequential([
    Flatten(),
    Dense(units=128, activation='relu'),
    Dense(units=10, activation='softmax')
])
```

Anschließend können die alten Gewichte mit dem neuen Modell geladen werden. Hierfür wird die `load\_weights` Funktion von Keras benutzt.

```
model3.load_weights('my_weights.h5')
```

4 Convolutional Layer

Besonders bei der Bilderkennung sind Covolutional Layer sehr nützlich, denn sie sind besonders hilfreich, die für das Modell interessanten Teile eines Bildes von dem Hintergrund und uninteressanten Teilen zu trennen. Dieses Kapitel behandelt die Grundlagen von Convolutional Layern und erklärt dabei anhand von einfachen Beispielen die wichtigsten Begriffe und Konzepte.

4.1 Grundlegende Begriffe

Eine Convolutional-Layer ist eine Ebene, an der die Eingabe-Daten gefaltet werden. Sie ist in der Lage, in den Eingabe-Daten Strukturen, Merkmale und Muster zu erkennen. Diese können beispielsweise gerade Linien, Kurven, Kreuze oder andere Formen sein.

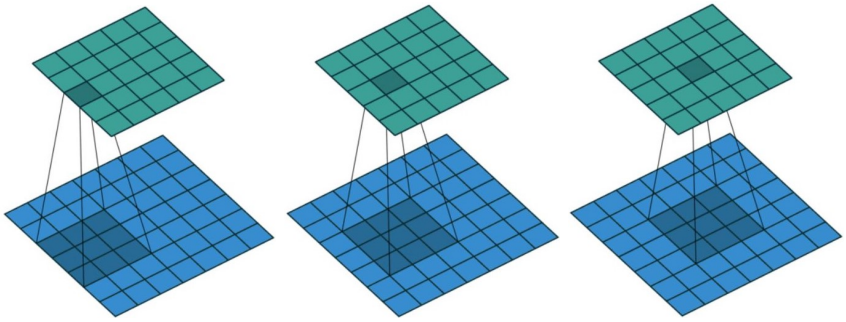
- Die Faltung der Eingabe-Daten erfolgt mit einer Matrix, die auch als **Kernel** bezeichnet wird und eine bei der Erstellung des Netzwerks festgelegte Größe (zum Beispiel 3x3) besitzt.

Der Code für die Erstellung einer Conv2D-Layer kann folgende Form haben:

```
model = Sequential()
model.add(Conv2D (1,(3,3),
            input_shape=(7,7,1)))
```

Diese Convolutional-Layer hat nur einen Filter mit einem Kernel der Größe 3x3. In Abbildung 5 ist zu sehen, wie sich der Kernel Schritt für Schritt über die Eingabe-Daten, hier die blaue 7x7-Matrix, bewegt. Dabei werden die 9 Werte aus der 3x3 Eingabe-Matrix mit den 9 Werten des Kernels multipliziert und anschließend summiert. Das Ergebnis wird dann abgespeichert und in einer neuen Matrix, hier die grüne 5x5-Matrix, abgespeichert. Diese ist dann die Ausgabe dieses Kernels.

- Der Parameter **strides** bestimmt die Schrittgröße mit der sich der Kernel über die Eingabe-Daten bewegt. Der Standard-Wert ist 1. Mit einem größeren Wert können die Größe der Ausgabe-Matrizen verringert und potentiell redundante Informationen verhindert werden.
- Mit dem Parameter **padding** kann festgelegt werden, ob der Kernel auch über den Rand hinaus gehen kann. Dies ist dann nützlich, wenn die Eingabe-Daten nicht verkleinert werden sollen. Im Beispiel einer 5x5



(a) 1. Schritt (b) 2. Schritt (c) 3. Schritt

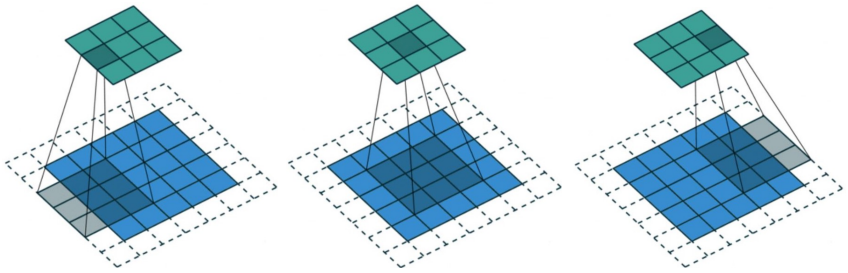
Figure 5: Funktionsweise des Kernels in einer Convolutionl-Layer.

Eingabe-Matrix und eines 3x3-Kernels würde der Kernel ohne padding an 9 Positionen Rechnungen durchführen, mit padding jedoch an 25 Positionen. Für den Parameter sind die Werte `valid` und `same` und `casual` möglich, wobei ersterer die Standard-Belegung ist.

Die Erstellung einer Convolutional-Layer mit strides und padding kann folgendermaßen aussehen:

```
model = Sequential()
model.add(Conv2D(1,(3,3),
                 strides=2,
                 padding='same',
                 input_shape=(7,7,1)))
```

Hier bewegt sich der Kernel im Vergleich zum vorherigen Beispiel mit der doppelten Schrittweite. Mit dem **padding**-Parameter festgelegt, kann der Kernel nun die Rand-Werte besser berücksichtigen. Wie sich der Kernel der neuen Faltungs-Ebene über die Eingabe-Daten bewegt ist in Abbildung 6 zu sehen.



(a) 1. Schritt (b) 2. Schritt (c) 3. Schritt

Figure 6: Funktionsweise des Kernels in einer Convolutionl-Layer mit strides=2 und padding='same'.

4.2 Ein einfaches Beispiel

In diesem Abschnitt wird, ausgehend von dem Beispiel aus Kapitel 3, ein neues Netzwerk für die Erkennung von handschriftlich geschriebenen Ziffern, diesmal mit Convolutional Layern, erzeugt. Der gesamte Python-Code ist in der Datei `conv_network.py` zu finden.

Conv2D-Layer funktionieren mit der Eingabe von zweidimensionalen Daten, diese sind in unserem Fall die Bilder der Ziffern. Die Bilder des MNIST Datensatzes haben die Form (28,28), allerdings erwarten die Conv2D-Layer Daten mit einer dritten Dimension, welche bei Bildern den Farbkanal darstellen kann. Es ist also notwendig, die Trainings- und Test-Daten zunächst in ein dreidimensionales Numpy-Array umzuwandeln und somit in die Form (28,28,1) zu bringen:

```
train_samples = train_samples.reshape(len(train_samples),28,28,1)
test_samples  = test_samples.reshape(len(test_samples),28,28,1)
```

Anschließend kann ein neues Modell erstellt werden.

```
model = Sequential([
    Conv2D(filters=32, kernel_size=(3,3), activation='relu',
            input_shape=(28,28,1)),
    Conv2D(filters=64, kernel_size=(3,3), activation='relu'),
    MaxPool2D(pool_size=(2,2)),
    Dropout(0.25),
    Flatten(),
    Dense(units=128, activation='relu'),
    Dropout(0.5),
    Dense(units=10, activation='softmax')
])
```

Dieses hat diesmal zwei Convolutional Layer mit 32 und 64 Neuronen. Hinter diesen beiden Layern wird eine MaxPool2D-Layer eingebaut, welche Overfitting

reduzieren und die Komplexität des Modells verringern soll. Die gleiche Funktion erfüllt die Dropout-Layer. Hier werden 25% der Neuronen zufällig auf Null gestzt. Flatten reduziert den Inpurt der nächsten Layer auf eine Dimension, indem der Output der letzten Layer in ein ein-dimensionalen Array transformiert wird. Anschließend wird eine Dense-Layer mit 128 Neuronen sowie eine weiterer Dropout Layer von 50% eingefügt. Zum Schluss kommt wieder eine Dense-Layer mit 10 Neuronen, welche die Klassifikation der 10 Ziffern vornimmt.

Auch dieses Modell muss compiliert werden. Dazu wird der Optimizer Adadelta genutzt werden. Durch eine Vergrößerung der Learning-Rate von dem Standard-Wert 0.01 auf 0.1 kann ein gutes Ergebnis erreicht werden, wobei gleichzeitig ein erheblich reduzierter Rechenaufwand benötigt wird.

```
model.compile(optimizer=keras.optimizers.Adadelta(learning_rate=0.1),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Anschließend kann das Modell trainiert werden. Dies funktioniert wieder wie in Abschnitt 3.5, diesmal allerdings mit nur 10 Epochen. Der Prozess des Trainings kann in den Abbildungen 7 und 8 gesehen werden. Hier sind die Kurven der Treffergenauigkeiten und Verlust-Funktionen für Trainings- und Validation-Daten dargestellt. Zu sehen ist, dass der Verlauf der Loss-Funktion der Validation-Daten im Vergleich zu Abbildung 3 deutlich abgeflacht ist und sich nicht mehr von den Trainingsdaten entfernt. Auch die accuracy der Validation-Daten nähert sich den Trainings-Daten nun ausschließlich an. Klare Zeichen dafür, dass es nun kein Overfitting mehr gibt.

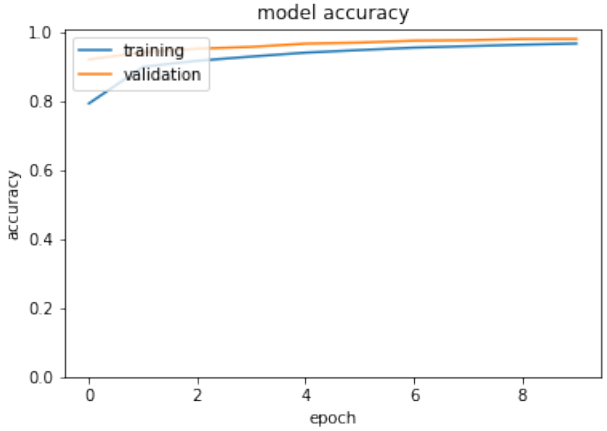


Figure 7: Kurve der Treffergenauigkeit des Modells mit Convolutional Layer für Trainings- und Validation-Daten.

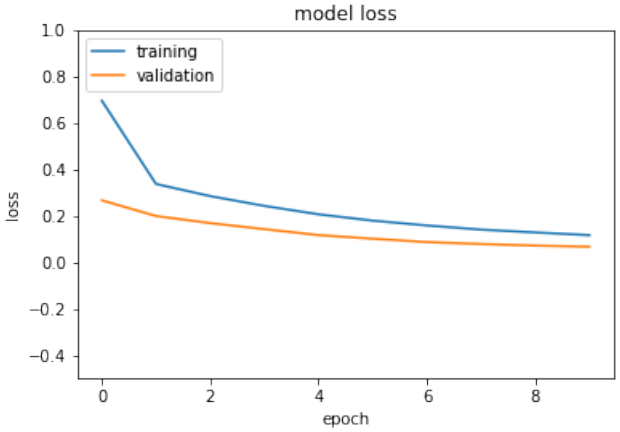


Figure 8: Kurve der Loss-Funktion des Modells mit Convolutional Layer für Trainings- und Validation-Daten.

5 Starthilfe: HPC Rechencluster

Dieses Kapitel gibt einen Überblick über den Aufbau das Hummel Rechenclusters und die Verwendung der Rechenkapazitäten in Hinblick auf Machine Learning. Eine umfangreiche Dokumentation findet sich auf der ofiziellen Website.

5.1 Zugang und erste Schritte

Um von außerhalt des Uni Netzwerks mit dem HPC Rechenzentrum arbeiten zu können, muss zunächst eine Verbindung mit dem VPN der Universität Hamburg hergestellt werden. Eine Anleitung dazu findet sich auf der Uni-Hamburg Website. Im zweiten Schritt muss die Freischaltung von dem Support hpc@uni-hamburg.de vorgenommen werden. Dies erfolgt per Austausch eines öffentlichen ssh Schlüssels. Ein neues Schlüsselpaar vom Typ RSA kann mit den folgenden Befehlen erstellt und verwendet werden. Die Schlüssellänge sollte nicht unter 4096 bits betragen.

```
ssh-keygen -t rsa -b 4096 -f $HOME/.ssh/id_rsa_hummel
ssh -i $HOME/.ssh/id_rsa_hummel
```

Nun kann die Verbindung zum Rechenzentrum von der Konsole aus per secure shell (ssh) hergestellt werden. Es sind zwei login-Gateways verfügbar.

```
hummel1.rrz.uni-hamburg.de
hummel2.rrz.uni-hamburg.de
```

Beides sind lediglich Gateways und nur für den login zu verwenden. Die beiden Ordner $WORK und $HOME sind jedoch verfügbar, sodass Daten per "secure copy" (scp) kopiert werden können. Auf den Front-End-Knoten kann sich von den Login-Knoten per ssh verbunden werden.

```
ssh front1
ssh front2
```

Beide Schritte können in einem Befehl zusammengefasst werden, wobei das -t für ein interaktives Terminal sorgt.

```
ssh -t Stine-Kennung@hummel1.rrz.uni-hamburg.de ssh front1
```

5.2 Installation von Anaconda und Tensorflow

Um Anaconda und Tensorflow auf den Compute Clustern des HPC Rechenzentrums benutzen zu können, muss zunächst der Anaconda3-Installer von der Anaconda Website heruntergeladen werden. Um die Installation durchführen zu können, muss der Installer in den persönlichen $WORK-Ordner kopiert werden. Dafür bietet sich SCP (Secure Copy Protocol) an. Zuerst wird der Pfad zur zu kopierenden Datei angegeben. Dahinter muss die Adresse des Rechenclusters sowie der Pfad zum $WORK Ordner angegeben werden. Bei Problemen mit der ssh-key Authentifizierung kann es helfen, nach dem „scp“ per „-i /.ssh/ssh-key“ auf die entsprechende shh Identität zu verweisen.

```
scp home/Pfad/zu/Anaconda3-2021.05-Linux-x86_64.sh
    Stine-Kennung@hummel1.rrz.uni-hamburg.de:/work/Stine-Kennung/
```

Nun kann die eigentliche Installation von Anaconda gestartet werden. Dazu kann die entsprechende bash Datei mit dem Befehl

```
bash Anaconda3-2021.05-Linux-x86_64.sh
```

ausgeführt werden. Zu Beginn muss die Lizenz-Vereinbarung akzeptiert werden. Danach läuft die Installation automatisch bis zum Ende. Der Pfad zur ausführbaren Datei conda.sh sollte anschließend in der Datei .bashrc im home Ordner eingetragen werden. Dazu kann die .bashrc Datei beispielsweise mit VIM geöffnet und bearbeitet werden. Der Eintrag könnte dann wie folgend aussehen.

```
/work/Stine-Kennung/anaconda3/etc/profile.d/conda.sh
```

Nun muss die Installation aktiviert werden.

```
source ~/.bashrc
```

Danach kann überprüft werden, ob die Installation erfolgreich war. Mit dem folgenden Befehl werden alle Pakete der aktuellen Anaconda Umgebung angezeigt.

```
conda list
```

War das erfolgreich, kann optional eine neue Anaconda Umgebung eingerichtet werden. In diesem Fall wird eine neue Python3 Umgebung mit dem Namen my_env erstellt und anschließend aktiviert.

```
conda create --name my_env python=3
conda activate my_env
```

Bei Bedarf kann nun der conda Befehl benutzt werden, um TensorFlow mit GPU-Support und matplotlib zu installieren.

```
conda install tensorflow-gpu
conda install matplotlib
```

Die Installation ist nun abgeschlossen. Wurde die Sitzung geschlossen, können Anaconda und die `my_env` Umgebung mit den Befehlen

```
source ~/.bashrc
conda activate my_env
```

wieder aktiviert werden.

5.3 Batch-Verarbeitung

Um einen Compute-Job auf einem der Rechenknoten ausführen zu lassen, muss das Batch-System verstanden werden. Auf dem Hummel wird das Batch-System SLURM (Simple Linux Utility for Resource Management) verwendet. Eine umfangreiche Dokumentation findet sich auf der Slurm Website.

5.3.1 Beispieljob

Um einen Job zu erstellen, wird ein Job-Skript benötigt. Dies ist nur ein Bash-Skript mit besonderen Befehlen für die Job-Durchführung in den obersten Zeilen. Hier soll zunächst ein Job erstellt werden, der die Datei run_example.py ausführt. Das zugehörige Job-Skript keras_example.sh kann dann wie folgt aussehen.

```
#!/bin/bash
# Hier werden die Job-Parameter festgelegt:
#SBATCH --job-name=keras_example
#SBATCH --partition=gpu
# Die gewünschte Partition. Hier der GPU-Knoten.
#SBATCH --nodes=1
#SBATCH --tasks-per-node=16
#SBATCH --time=00:10:00
# Zeitlimit nicht vergessen!
#SBATCH --export=NONE
#SBATCH --output=keras_example_output
#SBATCH --error=keras_example_error
#SBATCH  --mail-user=name@studium.uni-hamburg.de
# Mail-Adresse muss geändert werden!
#SBATCH --mail-type=ALL
# Alternativ: BEGIN, END, FAIL

# Abbruch bei erstem Fehler.
set -e

# Hier wird zunächst die benötigte Umgebung inklusive aller Module geladen.
module switch env env/2020Q3-gcc-openmpi
source ~/.bashrc
conda activate my_env
module load cuda

# Hier beginnt die eigentliche Arbeitsanweisung.
python $HOME/scripts/run_example.py
```

In die oberste Zeile kommt, wie bei jedem bash-Skript, der Hash-Bang. Darunter werden die Job-Parameter mit Kommentaren und dem Wort „SBATcH“ festgelegt. Der Job-Name ist in diesem Fall keras_example. Dieser ist öffentlich und bei der Auflistung aller laufenden Jobs zu sehen. Als Partition wurde hier der GPU-Koten gewählt. Andere verfügbare Partitionen sind, sofern freigeschaltet, der Standard-Knoten `std`, der große Knoten `big`, der Spezial-Knoten `spc` und die Default-Partition `all`, welcher alle Knoten der Partitionen `std` und `big` beinhaltet. Anschließend kann die Anzahl der Rechen-Knoten, die Anzahl der Tasks pro Knoten und die maximale Durchführungsdauer des Jobs ausgewählt werden. Ebenfalls festgelegt werden die Namen der output und error Dateien,

welche während der Durchführung des Jobs im aktuellen Verzeichnis angelegt werden. Benachrichtigungen über den Beginn, das Ende oder einen Abbruch des Jobs können an eine Mail-Adresse versendet werden. Hier sollte möglichst eine Mail-Adresse der Universität Hamburg genutzt werden.

5.3.2 SLURM Befehle

Hier folgt eine Auflistung der wichtigsten SLURM Befehle zur Verwaltung von Jobs.

- Um ein Job-Skript hochzuladen:

  ```
  $ sbatch script.sh
  ```

- Um alle eigenen laufenden Jobs anzuzeigen:

  ```
  $ squeue -u $USER
  ```

  Dies gibt die Information in folgender Form aus:

  ```
  JOBID   PARTITION   NAME    USER    ST  TIME    NODES  NODELIST(REASON)
  1461514 std         name    abc     R   1:37    2      node269
  ```

- Um sich detaillierte Informationen zu einem Job anzeigen zu lassen:

  ```
  $ scontrol show job JOBID
  ```

- Um einen Job abzubrechen:

  ```
  $ scancel JOBID
  ```

- Falls es ein Problem mit dem Compute-Knoten gibt, ist die Standard-Konfiguration von SLURM den Job automatisch neu zu starten. Falls dieses Verhalten nicht gewünscht ist, kann die `--no-requeue` Option bei der Erstellung eines Jobs benutz werden:

  ```
  $ sbatch --no-requeue script.sh
  ```

  Alternativ kann der folgende Befehl im Job-Skript benutzt werden:

  ```
  #Sbatch --no-requeue
  ```

- Eine umfassendere Auflistung von SLURM-Befehlen ist hier zu finden.

5.4 Arbeiten mit CPU Knoten

5.4.1 CPU Partitionen

Bis auf die Spezial-Knoten sind alle Compute-Knoten mit 2 CPUs vom Typ Intel Xeon E5-2630v3 ausgestattet. Je CPU:

- 8 Rechenkerne
- Grundfrequenz 2,4 GHz
- L3-Cache 20 MByte
- Befehlssatzerweiterung AVX 2.0

Die für CPU-Jobs zu verwendenen Partitionen sind also:

- Die **Standard-Partition** `std` wird automatisch ausgewählt, wenn keine bestimmte andere Partition angefordert wird. Sie besteht aus 316 Knoten (`node1` bis `node2`) mit jeweils 64 GByte Hauptspeicher. Zwei dieser Knoten dienen als Anmelde-Knoten und sind mit `front1` und `front2` ansprechbar.
- Die **Große Partition** `big` besteht aus 24 Knoten (`node371` bis `node394`) mit jeweils 256 GByte Hauptspeicher.
- Die **Spezial-Partition** `spz` besteht aus 2 Knoten (`node395` und `node396`) mit jeweils 1024 GByte Hauptspeicher.

5.4.2 CPU Parallelisierung

5.5 Arbeiten mit GPU Knoten

5.5.1 GPU Partition

Die GPU-Partition `gpu` besteht aus 54 Knoten (`node 317` bis `node 370`), jeweils mit 64 GByte Hauptspeicher und einer NVIDIA K80 Dual-GPU ausgestattet:

- 4992 NVIDIA CUDA-Cores
- 24 GByte GDDR5 Speicher
- Speicher-Bandbreite von 480GByte/s
- Grundfrequenzt 562 MHz
- Boost-Frequenz 824 MHz

5.5.2 GPU Parallelisierung

Die Parallelisierung der GPUs wird mithilfe der Cuda-bewussten Bibliothek OpenMPI gesteuert. Um dies zu ermöglichen sollte das System der Environment-Modules Beachtung finden. Damit der geschriebene Code von einer der OpenMPI Bibliotheken compiliert werden kann, müssen zunächst die entsprechenden Umgebungen mit OpenMPI und dem Cuda-Modul geladen werden:

```
module switch env env/2020Q3-gcc-openmpi
module load cuda
```

6 Klassifizierung von SDSS Spektraldaten

6.1 Der SDSS Katalog

Die SLOAN DIGITAL SKY SURVEY (SDSS) ist ein in 2004 begonnenes Projekt, welches sich mit der Vermessung des Himmels bei 5 Wellenlängenbereichen beschäftig. Dafür wurde eigens ein Telescop mit 2,5 m Hauptspiegeldurchmesser am Apache Point Observatory in New Mexiko gebaut.
Der zwölfte Daten-Release beinhaltet mehr als vier Millionen Spektren von Galaxien, Quasaren und Sternen in den Wellenlängenbereichen u, r, g, i und z. Der Zugriff auf die Daten kann beispielsweise über den Science Archive Server (SAS) oder den Skyserver erfolgen.

6.2 DR12 Science Archive Server (SAS)

Auf die Daten des *Science Archive Servers* des zwölften Daten-Release lassen sich über die Advanced Optical Spectra Search zugreifen. Über eine Suchfunktion können dort die gewünschten Spektren gefunden und als Fits-Dateien heruntergeladen werden. In mehreren Reitern können verschiedene Parameter eingegrenzt werden. Die wichtigsten Parameter sind die Identifikationsnummer der Platte (`Plate ID`), das Julianische Datum (`MJD`), die Faser der gewählten Platte (`fiber`),

die Identifikationsnummer des beobachteten Objekts (`Thing_ID`) und die maximale Anzahl an Suchergebnissen. Der letzte Wert ist auf 2000 Ergebnisse begrenzt. In anderen Reitern, wie z.B. bei *Class* oder *Sky Region*, kann die Klasse oder der Ort am Sternenhimmel des gewünschten Objekts eingegrenzt werden.

6.3 Download vom Skyserver

Der Skyserver erlaubt den Zugriff auf alle öffentlich verfügbaren Daten der Sloan Digital Sky Server (SDSS). Dies beinhaltet sowohl Bilder von Himmelsobjekten, gespeichert in .jpg und .fits Dateien, als auch als .jpg Dateien herunterladbare Spektren im Bereich von etwa 3800 bis 9200 $\mathring{A}$ sowie photometrische und spektroskopische Daten in diversen Datenformaten. Der Zugriff auf diese Daten ist über eine Vielzahl an Tools möglich. Einen besonders interaktiven Zugriff bietet das Navigations-Tool, welches den Benutzer durch Klicken und Zoomen den Nachthimmel erkunden lässt. Um den gesamten Datensatz umfangreich durchsuchen zu können, sollte sich allerdings mit der SQL-Suche auseinandersetzt werden.

Der Schema Browser listet alle Tabellen auf, die durchsucht werden können. Insbesondere finden sich dort Auflistungen und Erklärungen der in den Tabellen beinhalteten Spalten. Eine der dort aufgeführten Tabellen ist die *SpecObj* Tabelle. Diese beinhaltet die Spektren aller beobachteten Himmelsobjekte sowie eine große Zahl an zugehörigen Daten, wie die Objekt-Identifikationsnummer *targetObjID*, die Platten Identifikationsnummer *plateID* und das modifizierte Julianische Datum *mjd*, an welchem das Spektrum aufgezeichnet wurde.

Alle diese Daten können mit einer SQL-Suche durchsucht werden. Eine einfache Suche wäre beispielsweise die Folgende:

```
SELECT top 10 ra, dec, targetObjID, class
FROM SpecObj WHERE ra < 10
AND ra > 5 and class = 'star'
```

Mit dieser Suche werden Rektaszension, Deklination, targetObjID und Klasse der ersten 10 Treffer ausgegeben, deren Rektaszension zwischen 5° und 10° liegt und deren Klasse in die der Sterne fällt. Die erhaltenen Informationen können dann genutzt werden, um auf der Bulk Imaging Search die gewünschten Spektren herunterzuladen. Alternativ können beide Schritte in einem Python-Skript verbunden werden.

Um zu verstehen wie das funktioniert, soll im Folgenden eine Datenbank mit Spektren im .fits-Format von jeweils 1000 Sternen, Galaxien, Quasaren (QSO) und aktiven Galaxiekernen (AGN) angelegt werden. Das folgende python-Skript *download_sdss_fits.py* erfüllt diese Aufgabe.

```
import numpy as np
import sys
import os
import subprocess
import astropy.io.fits as pyfits
from astroquery.sdss import SDSS
from astropy.coordinates import SkyCoord, ICRS
import astropy.units as u
import requests

sdss_path = 'https://data.sdss.org/sas/dr16/sdss/spectro/redux/26/spectra/'

query1  =   "SELECT top 1000 plate, mjd, min(fiberid) as fiberid, class FROM SpecObj
            WHERE class = 'star' GROUP BY plate, mjd, class ORDER BY plate, mjd, class"
query2  =   "SELECT top 1000 plate, mjd, min(fiberid) as fiberid, class FROM SpecObj
            WHERE class = 'galaxy' AND   subClass != 'AGN' GROUP BY plate, mjd, class
            ORDER BY plate, mjd, class"
query3  =   "SELECT top 1000 plate, mjd, min(fiberid) as fiberid, class FROM SpecObj
            WHERE class = 'QSO' AND   subClass != 'AGN' GROUP BY plate, mjd, class
            ORDER BY plate, mjd, class"
query4  =   "SELECT top 1000 plate, mjd, min(fiberid) as fiberid, class FROM SpecObj
            WHERE subClass = 'AGN' GROUP BY plate, mjd, class
            ORDER BY plate, mjd, class"

class_names = ['star','galaxy', 'QSO', 'AGN']
queries = [query1, query2, query3, query4]

for i in range(4):
    sdss = SDSS.query_sql(queries[i])
    speclist  = open('speclist.txt', 'w')

    for plate, mjd, fiberid in zip(sdss['plate'],sdss['mjd'],sdss['fiberid']):
        speclist.write("%04d/spec-%04d-%d-%04d.fits \n" %(plate, plate, mjd, fiberid))
    speclist.close()

    with open('speclist.txt', 'r') as f:
        names = f.readlines()

    for item in names:
        name = item[:-2]
        url = sdss_path + name
        r = requests.get(url)
        target_file = 'F:\data\spectral_fits\\' + class_names[i] + '\\' + name[5:]

        with open(target_file,'wb') as f:
            f.write(r.content)
```

In den ersten Zeilen werden alle benötigten Pakete importiert. Falls noch nicht vorhanden, müssen die Module *astroquery* und *requests* mit den Befehlen

```
conda install -c conda-forge astroquery
pip install requests
```

installiert werden. Astroquery ermöglicht den Zugriff auf den Skyserver und requests organisiert den Download. Falls das Skript auf Linux ausgeführt werden soll, würde sich auch das *wget* Modul anbieten. Anschließend wird die Adresse zum Speicherort der SDSS Spektraldaten in der Variable `sdss_path` abgespeichert.

Um alle 4000 Spektren zu finden, werden insgesamt 4 SQL-Suchen durchgeführt. Diese werden in 4 Variablen abgespeichert, über die später iteriert werden kann. In den jeweiligen SQL-Suchen werden die Objekt-Identifikationsnummer `targetObjID`, die Platten Identifikationsnummer `plate`, das modifizierte Julianische Datum `mjd`, die Faser Identifikationsnummer `fiberid` und die Bezeichnung der Klasse `class` abgefragt. Da die aktiven Galaxiekerne (AGN) eine Subklasse der Galaxien und Quasare darstellen, muss bei den jeweiligen Suchen sichergegangen werden, dass die ausgegebenen Galaxien bzw. Quasare keine AGNs sind.

Unterhalb der SQL-queries werden die Klassennamen und queries in zwei Listen abgespeichert. Über diese Listen wird in der darunterliegenden for-Schleife iteriert. Dabei wird benutzt, dass alle in .fits-Dateien abgespeicherten Spektren nach dem folgenden Schema bezeichnet wurden:

```
plate/spec-plate-mjd-fiberid.fits
```

Indem die jeweiligen Parameter `plate`, `mjd` und `fiberid` mit den aus der SQL-Suche erhaltenen Parametern ausgefüllt werden, können die Speicherorte der gewünschten Spektren ausgewählt werden. Diese können schlussendlich mit dem `request` bzw. `wget` Befehl heruntergeladen werden.

6.4 Spektren in Numpy Arrays abspeichern

Um die 4000 in .fits-Dateien abgespeicherten Spektren möglichst zugänglich abzuspeichern, bieten sich .npy-Dateien an. Die Daten können in numpy-Arrays eingelesen und in externen Dateien abgespeichert werden. Der Vorteil ist, dass das Einlesen von Daten aus 1-3 numpy-Arrays deutlich schneller und unkoplizierter ist, als jedes Mal über 4000 .fits-Dateien zu iterieren.

Im Beispiel wurden Die Helligkeits-Werte, Labels und Wellenlängen der 4000 Spektren in den drei .npy-Dateien `data.npy`, `labels.npy` und `wavelengths.npy` abgespeichert. Das zugehörige Skript kann wie gewöhnlich im GitHub in der Datei `fits_to_npy.py` gefunden werden, hier nur eine kurze Erklärung: Die Helligkeitsdaten werden in einem numpy-Array der Form (4000, 3522) gespeichert. Dieses Array enthält also 4000 Elementen mit je 3522 Einträgen - also zu jedem Spektrum die Helligkeitswerte bei allen 3522 Wellenlängen. Die Wellenlängen

bei denen jeweils gemessen wurde, sind in dem Numpy-Array `wavelengths` abgespeichert. Dieses hat die Form (3522,), enthält also nur einmal 3522 Einträge. Mehr ist nicht nötig, da alle Spektren Messwerte an den identischen Wellenlängen besitzen. Schlussendlich enthält das numpy-Array `labels` die labels aller 4000 Spektren und hat somit die Form (4000,). Die labels 0, 1, 2 und 3 bezeichnen dabei die Klassen AGN, galaxy, QSO und stars in alphabetischer Reihenfolge.

6.5 Spectral Classifier erstellen

In diesem Abschnitt wir erklärt, wie das Convolutional Network zur Klassifizierung der Spektraldaten erstellt und trainiert werden kann. Das zugehörige Skript kann im GitHub unter dem Namen `spectral_classifier.py` gefunden werden.

Zunächst müssen die Spektral-Daten, labels und Wellenlängen in die drei numpy-Arrays `data`, `labels` und `wavelengths` geladen werden.

```
data = np.load(data_path + "data.npy")
labels = np.load(data_path + "labels.npy")
wavelengths = np.load(data_path + "wavelengths.npy")
```

Damit die Spektren später noch identifiziert werden können, wird die Liste `numbers` erstellt, welche den Spektren die Zahlen 0-3999 zuordnet.

```
numbers = range(4*samples_per_class)
```

Anschließend kann der Datensatz gemischt werden. Dabei muss beachtet werden, dass zueinander gehörende Daten, labels und Nummern an den gleichen Positionen bleiben. Dazu werden die Arrays und Listen zunächst zusammengefügt, gemischt und anschließend wieder voneinander getrennt.

```
z = list(zip(data, labels, numbers))
random.shuffle(z)
data_shuffled, labels_shuffled, numbers_shuffled = zip(*z)
```

Nun kann der Datensatz in einen Trainings- und einen Testdatensatz aufgeteilt werden. Hier wurde sich für ein Verhältnis von 9:1 entschieden.

```
data_training = np.asarray(data_shuffled[:split_index])
data_test = np.asarray(data_shuffled[split_index:])

labels_training = np.asarray(labels_shuffled[:split_index])
labels_test = np.asarray(labels_shuffled[split_index:])

numbers_training = numbers_shuffled[:split_index]
numbers_test = numbers_shuffled[split_index:]
```

Damit die Daten an eine Convolutional Layer übergeben werden können, muss der Datensatz in die richtige Form gebracht werden. Dies geht mit der `numpy.reshape()` Funktion.

```
input_shape = (3522,1)
data_training_r = np.reshape(data_training, newshape=(len(data_training),
        input_shape[0], input_shape[1]))
data_test_r  = np.reshape(data_test, newshape=(len(data_test),
        input_shape[0], input_shape[1]))
```

Nun kann das eigentliche Netzwerk erstellt werden. Eine relativ gut funktionierende Struktur besteht aus folgenden Teilen: Zunächst einer ersten Convolutional-Layer mit 64 Filtern, einer kernel-Größe von 80 und mit 10er strides. Nach einer MaxPooling- und Dropout-Layer folgt eine zweite Convolutional-Layer mit diesmal 128 Filtern, einer kernel-Größe von 128 und 10er strides. Nach einer weiteren MaxPooling- und Dropout- Layer folgt eine Flatten Layer, welche die Daten für die darauf folgende Dense-Layer vorbereitet. Diese besteht aus 128 Neuronen. Nach einer weiteren Dropout-Layer folgt lediglich eine letzte Dense-Layer mit 4 Neuronen, welche den Output des Netzwerks darstellen soll.

```
model = Sequential([
    Conv1D(filters=64, kernel_size=80, strides=10,
            activation='relu', input_shape=(3522,1)),
    MaxPooling1D(3),
    Dropout(0.35),
    Conv1D(filters=128, kernel_size=40, strides=10, activation='relu'),
    MaxPooling1D(3),
    Dropout(0.35),
    Flatten(),
    Dense(units=128, activation='relu'),
    Dropout(0.35),
    Dense(units=4, activation='softmax')
])
```

Anschließend kann das Netzwerk compiliert und trainiert werden. Als optimizer wurde `Adam` und als compiler `sparse_categorical_crossentropy` gewählt. Die `epochs`, `batch_size` und der `validation_split` können natürlich nach Belieben variiert werden. Die Trainings- und Testdaten werden zusätzlich auf Werte zwischen 0 und 1 normiert. Dadurch kann das Training erheblich verbessert und sogar Overfitting reduziert werden.

```
model.compile(optimizer='Adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

x_train = tf.keras.utils.normalize(data_training_r, axis=1)
x_test = tf.keras.utils.normalize(data_test_r, axis=1)

y_train = labels_training
y_test = labels_test

history = model.fit(x_train, y_train,
```

```
epochs=75, validation_split=0.1,
shuffle=True, batch_size=200,
verbose=1)
```

Ein Trainingsdurchlauf mit 75 Epochen, einem `validation_split` von 0.1 und einer `batch_size` von 200 lieferte eine Trefergenauigkeit von etwa 80%. Der Verlauf von Treffergenauigkeit und loss-Funktion ist in den Abbildungen 9 und 10 zu sehen.

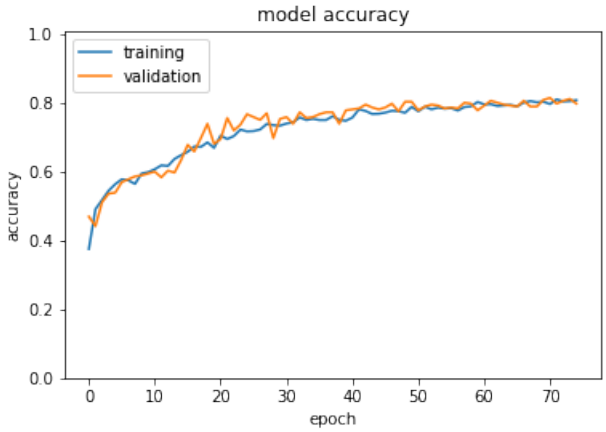


Figure 9: Kurve der Treffergenauigkeit des Netzwerks zur Klassifizierung von Spektren mit Trainings- und Validation-Daten.

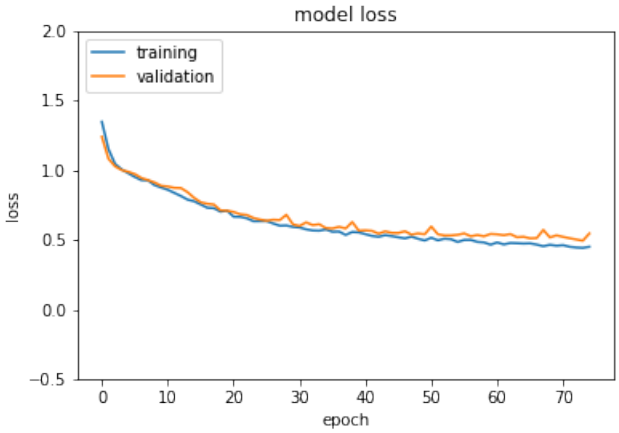


Figure 10: Kurve der Loss-Funktion des Netzwerks zur Klassifizierung von Spektren mit Trainings- und Validation-Daten.

6.6 zWarning Flags

Ein nicht unrelevanter Teil der Spektren aus unserem Datensatz besteht aus Daten, die nicht repräsentativ für Spektren dieser Klasse sind. Dies sind zum Beispiel Spektren, die lediglich um die 0 herum schwanken, was normalerweise als Detektor-Rauschen eingestuft werden kann. Andere Spektral-Verläufe sind deutlich breiter als sonst üblich. Eine Untersuchung hat gezeigt, dass knapp die Hälfte der falsch klassifizierten Spektren solche Probleme haben.

Alle Spektren der SpecObj-Tabelle haben einen *zWarning*-Flag zugeordnet. Spektren mit einen zWarning-Flag gleich null haben keine bekannten Probleme. Ist der zWarning-Flag von null verschieden, so lässt sich dies auf folgende Fehler zurückführen:

Table 1: zWarning-Flags der SpecObj-Tabelle aus dem SDSS Datensatz.

zWarning-Flag	Name	Bedeutung
0	OK	Keine bekannten Probleme
1	LITTLE_COVERAGE	Zu kleine Wellenlängen Abdeckung
2	SMALL_DELTA_CHI2	Chi-Quadrat des besten Fits istzu nah an dem des zweitbesten Fits
3	NEGATIVE_MODEL	Synthetische Spektrum ist negativ (nur für stars und QSO)
4	MANY_OUTLIERS	Teil der Messpunkte mehr als 5 sigma weg vom besten Modell
5	Z_FITLIMIT	Chi-Quadrat Minimum an der Kante der redshift fitting range
6	NEGATIVE_EMISSION	Eine QSO-Linie exponiert negative Strahlung
7	UNPLUGGED	Die Detektor-Faser war während des Messvorgangs nicht eingesteckt

6.7 Goldener Datensatz

Um zu sehen, welchen Einfluss die mit einem zWarning-Flag gekennzeichneten Spektren auf die Funktion des Netzwerks haben, kann ein neuer, "goldener" Datensatz erzeugt werden. Dieser Datensatz soll lediglich Spektren mit zWarning=0 beinhalten. Dazu muss in der SQL-Abfrage beim Download der Spektren die Bedingung `WHERE zWarning=0` festgehalten werden. Ein Training mit diesem Datensatz führt zu einem Netzwerk mit einer um etwa 5-10% erhöhten Treffergenauigkeit. Die zugehörigen Kurven der Treffergenauigkeit und Loss-Funktion können in den Abbildungen 11 und 12 gefunden werden.

References

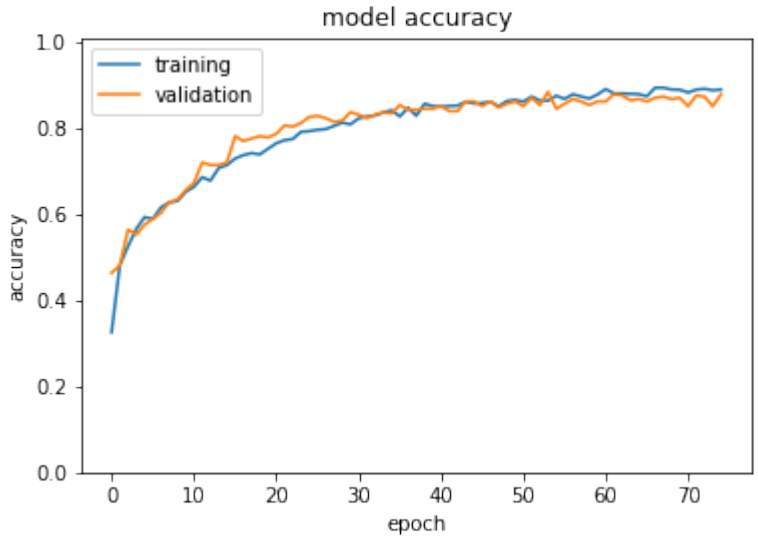


Figure 11: Kurve der Treffergenauigkeit des Netzwerks zur Klassifizierung von Spektren mit Trainings- und Validation-Daten mit nur zWarning=0 geflagten Daten.

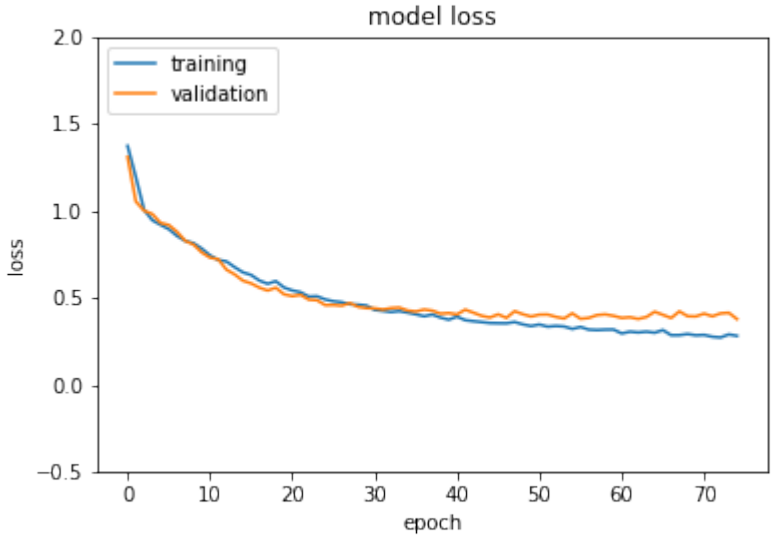


Figure 12: Kurve der Loss-Funktion des Netzwerks zur Klassifizierung von Spektren mit Trainings- und Validation-Daten mit nur zWarning=0 geflagten Daten.