Computer-Linguistische Anwendungen

CLA | B.Sc. | LMU

Vorlesung: Neuronale Netze

Philipp Wicke, PhD
Centrum für Sprach- und Informationsverarbeitung
Ludwig-Maximilians-Universität München
pwicke@cis.lmu.de

Keras

Übersicht und Anwendung

Was ist Keras?

- Bibliothek für Neuronale Netze in Python
- Einfache Verwendung der **gängigsten** Netzwerk Bestandteile und Lernverfahren
- Erweiterbar für **besondere** Verwendungen

Vorteile:

- Default Parameter (z.B. Initialisierung der Gewichte-Matrizen) sind gut
- Wenig Redundanz. Keras erkennt Zusammenhänge (z.B. Ein/Ausgabe-Dimensionen) automatisch, wenn möglich.
- Modular erweiterbar

Keras

Installation

```
pip3 install keras

# or

git clone https://github.com/keras-team/keras
cd keras
python3 setup.py install

# or

conda install keras
```

Keras

Auswahl der Backends

- Keras verwendet für Berechnungen andere Deep-Learning Bibliotheken, welche selbst ausgewählt werden können (*Tensorflow*, *Pytorch*, *Theano*, etc.)
- Empfehlung: *tensorflow* oder *pytorch*
- Um das Backend zu ändern kann eine Systemvariable gesetzt werden: `KERAS_BACKEND=tensorflow`
- Config. Datei um das Backend permanent zu ändern:

~/.keras/keras.json

```
{
"floatx": "float32",
"image_dim_ordering": "tf",
"epsilon": 1e-07,
"backend": "tensorflow"
}
```

Keras

Grundprinzip

```
from keras.models import Sequential
from keras.layers import SomeLayer, OtherLayer
model = Sequential()
model.add(SomeLayer(...))
model.add(OtherLayer(...))
model.add(...)
model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train)
```

`Sequential()` erzeugt ein Modell, in dem Layers sequenziell hintereinander geschaltet werden können.

- Für jede Layer wird zunächst das entsprechende Objekt erzeugt, und dieses zum Modell hinzugefügt
- Die jeweiligen Layers übernehmen die Ausgabe der vorhergehenden Layers als ihre Eingabe

Keras

Grundprinzip

```
from keras.models import Sequential
from keras.layers import SomeLayer, OtherLayer
model = Sequential()
model.add(SomeLayer(...))
model.add(OtherLayer(...))
model.add(...)
model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train)
```

`model.compile`: Wenn die Spezifikation des Modells abgeschlossen ist, wird dieses compiliert:

- Es wird spezifiziert, welcher Lernalgorithmus verwendet werden soll.
- Welche Kostenfunktion minimiert werden soll.
- Und welche zusätzlichen Metriken zur Evaluation berechnet werden sollen.

`model.fit`: Training (Anpassen der Parameter in allen Layers)

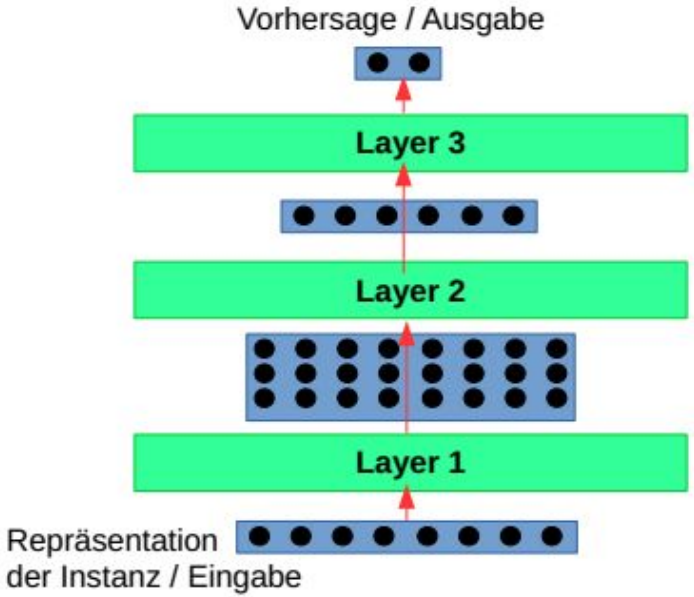
Keras

Embedding Layer

```
from keras.layers import Embedding
...
model.add(Embedding(input_dim=10000, output_dim=50))
...
```

Stellt Wortvektoren der Größe `input_dim` fur ein Vokabular der Größe `output_dim` bereit.

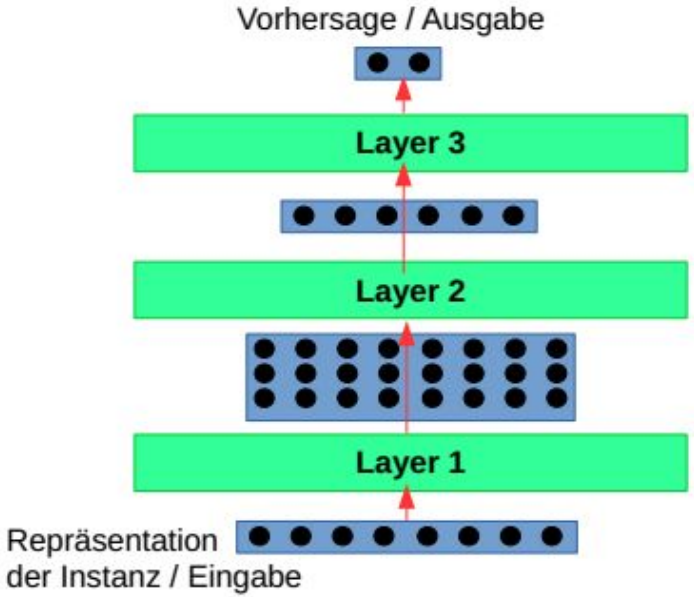
- Oft die erste Layer in einem Modell.
- Eingabe pro Instanz: Vektor mit Wort id's
- Ausgabe pro Instanz: Matrix; Sequenz von Wortvektoren



Keras

Embedding Layer

```
from keras.layers import Embedding
...
model.add(Embedding(input_dim=10000, output_dim=50, \
          weights=[word_vectors], trainable=False))
...
```



Die Parameter (Wortvektoren) der Embedding Layer

- ... können mit vortrainierten Vektoren (Word2Vec), oder zufällig initialisiert werden.
- ... bei vortrainierten Vektoren kann oft auf ein weiteres Optimieren der Wortvektoren verzichtet werden.

Keras

Embedding Layer

```
...
model.add(Embedding(input_dim = VOCAB_SIZE,
        output_dim = EMBEDDING_SIZE, mask_zero = True))
# output of embedding_layer is masked for timesteps where word id=0
...
```

Short sentence *pad pad pad*
1 1 0 0 0

This is a long sentence.
1 11 1 1

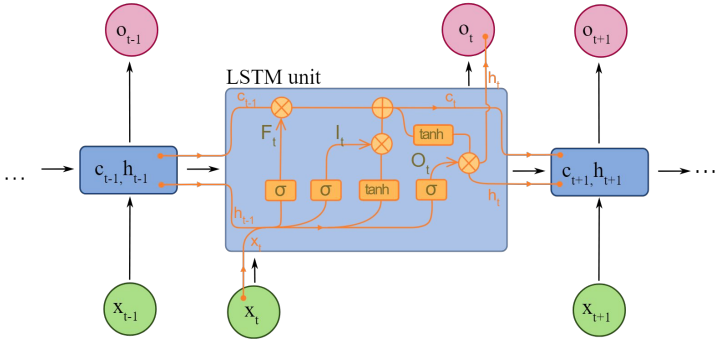
- **Masking** in Keras

  Keras erwartet als Eingaben (inputs) Numpy arrays.
  Listen verschiedener Länge (z.B. Satzrepräsentationen) können durch den Befehl:
  `pad_sequences(list_of_lists, max_length)`
  in ein Numpy-Array mit vorgegebener Spaltenanzahl umgewandelt werden.

Keras

RNN Layer + LSTM

```
from keras.layers import LSTM, Bidirectional
...
model.add(LSTM(units=100))
...
```



Mit der zu Beginn vorgestellten Variante des RNN können theoretisch zwar sehr komplexe Vorhersagen gelernt werden, die Parameter sind aber in der Praxis schwer zu optimieren (*vanishing gradient problem*).

Erweiterung des RNN, die das Optimieren der Parameter erleichtern, sind z.B. **LSTM** (long short-term memory network) und **GRU** (gated recurrent unit network)

Vanishing Gradient Problem:
Während jedem Trainings-Durchlauf werden die Gewichte über Backpropagation geupdated. Dieses Update ist proportional zu der partiellen Ableitung der Fehlerfunktion im Bezug auf das aktuelle Gewicht. Das Problem beschreibt nun, dass in manchen Fällen der Gradient verschwindend gering ist und nahezu keine Auswirkung auf das Gewicht mehr ausübt.

Long-Short-Term Memory Netzwerke umgehen dieses Problem in dem sie die Gradienten auch unverändert durch das Netzwerk propagieren lassen.

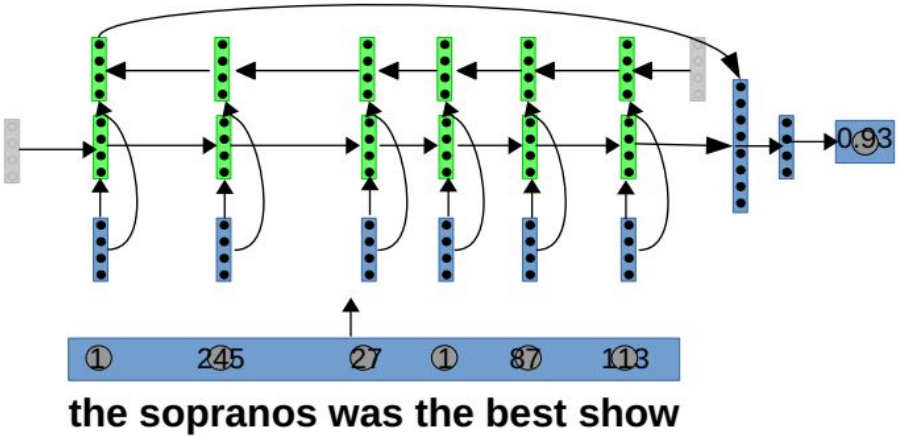
https://commons.wikimedia.org/wiki/File:Long_Short-Term_Memory.svg

Keras

RNN Layer

Zwei gegenläufige RNNs, Ausgabe sind die konkatenierten End-Vektoren (wie im Beispiel zuvor):

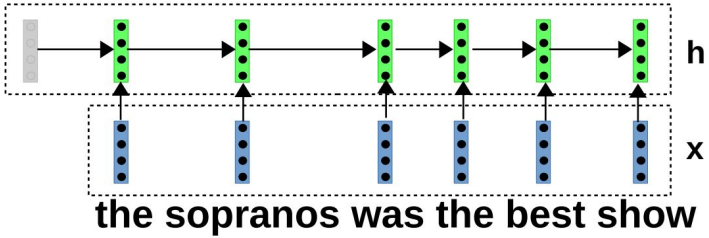
```
model.add(Bidirectional(LSTM(units=100)))
```



Keras

RNN Layer

Anstelle des Endvektors kann auch eine Matrix ausgegeben werden, die für jede Position den Zustandsvektor ***h*** enthält:




```
model.add(LSTM(units=100, return_sequences=True))
```

Für welche computerlinguistischen Aufgaben ist es nötig,
Zugriff auf die Zustandsvektor an jeder Position zu haben?

Immer, wenn eine Vorhersage für jede Position getroffen werden muss, z.B. Wortarten-Tagging.

Keras

Dense-Layer

Zwei Verwendungen:

Als Zwischen-Layer

- Kombiniert Information aus vorhergehender Layer.
- Nichtlinearität ist ***ReLu*** oder ***Tanh***.

```
from keras.layers import Dense
...
model.add(Dense(100, activation='tanh'))
…
```

Als Ausgabe-Layer

- Wahrscheinlichkeit einer Ausgabe.
- Nichtlinearit¨at ist Sigmoid (2 m¨ ogliche Klassen) oder Softmax (beliebig viele Klassen).

```
...
model.add(Dense(1, activation='sigmoid'))
...
```

Keras

Eingabe/Ausgabe-Dimensionen bei Satzklassifikation

```
from keras.layers import LSTM, Dense, Embedding
from keras.models import Sequential

VOCAB_SIZE, EMB_SIZE, HIDDEN_SIZE, NUM_TOPICS = 1000, 100, 200, 50
x = np.random.randint(size = (4, 80), low = 0, high = VOCAB_SIZE))

model = Sequential()
embedding_layer = Embedding(input_dim = VOCAB_SIZE, output_dim = EMB_SIZE)
model.add(embedding_layer)
print(model.predict(x).shape)
(4, 80, 100)
lstm_layer = LSTM(units = HIDDEN_SIZE)
model.add(lstm_layer)
print(model.predict(x).shape)
(4, 200)
output_layer = Dense(units = NUM_TOPICS, activation = "softmax"))
model.add(output_layer)
print(model.predict(x).shape)
(4, 50)
```

Vorhersage ohne Training um die Größe der resultierenden Vorhersage zu überprüfen

Hier: 4 Instanzen mit 80 Positionen mit jeweils einem Wort-Vektor der Größe 100 (Embedding)

Für die 4 Instanzen wird nun durch den LSTM Layer (mit hidden size 200) eine **Satzrepräsentation** gebildet. `return_sequences = True`liefert (4, 80, 200)

50 Themen, jede Klasse erhält einen Wert.

Keras

Training

- `fit` nimmt Tensoren X und Y als Argumente
- Die Shape muss den Erwarteten Eingabe und Ausgabe-Shapes entsprechen
- fit liefert ein history-Object mit loss-/metrics-Werten aller Epochen
- Per Default, shuffelt `fit` die Trainingsdaten

```
print(model.input_shape)
(None, None) # (batchsize, timesteps). None means that any size > 0 is okay.
print(model.output_shape)
(None, 50) # (batchsize, output_dim)
X, Y = # load_training_data()
print(X.shape)
(20, 30)
print(Y.shape)
(20, 50)
history = model.fit(X, Y, epochs = 5, shuffle = True)
print(history.history["loss"])
[0.317502856254577637, 0.26498502135276794, ...]
```

Keras

Training

Loss Funktionen:

- `binary_crossentropy` falls nur eine WK vorhergesagt wird (Sigmoid activation)
- `categorical_crossentropy` wenn WK-Verteilung über mehrere Klassen (Softmax activation)

Optimizer: `adam, rmsprop, sgd`

- Metrics: zusätzliche Fehlerfunktionen (neben dem Loss), die zur Analyse interessant sind

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

Keras

Training

`model.fit(...)`

- wendet einen Optimierungsalgorithmus an, um den Loss-Wert auf den Trainingsdaten zu minimieren (so, dass die Vorhersagen des Modells gut zu den Trainings-Labels passen)

- Hyper-parameter
  - `batch_size`: Für wieviele Instanzen ein Optimierungsschritt ausgeführt werden soll. (Optimierungsschritt ≠ Trainingsiteration)
  - `epochs`: Wieviele Trainingsiterationen ausgeführt werden sollen.

- `validation_data`: Tuple (`features_dev, labels_dev`)
  Entwicklungsdaten, z.B. um Trainingsfortschritt zu monitoren

Keras

Evaluation

```
X, Y = # load_dev_data()
results = model.evaluate(X, Y)
for name, number in zip(model.metrics_names, results):
        print(name, number)

loss 0.29085057973861694
acc 0.7510684013366699
```

```
X_train, Y_train = # load_train_data()
X_dev, Y_dev = # load_dev_data()
history = model.fit(X_train, Y_train, epochs = 5,
        validation_data = (X_dev, Y_dev))
# validation loss history in history.history["val_loss"]

#or use 10% of X_train, Y_train as validation set
history = model.fit(X_train, Y_train, epochs = 5, validation_split = 0.1)
```

Keras

Callbacks

- **`EarlyStopping`**: Stop training when a loss/metric stops improving
- **`ModelCheckpoint`**: Save model at regular intervals
- **`ReduceLROnPlateau`**: Reduce learning rate when loss stops improving
- …

```
from keras.callbacks import EarlyStopping, ModelCheckpoint

earlystop = EarlyStopping(monitor = "val_acc", patience = 5)
# stop training if validation accuracy did not improve for 5 epochs

checkpoint = ModelCheckpoint("./mymodel.h5",
        save_best_only = True, monitor = "val_acc")
# save model after epochs with improved validation accuracy improves

model.fit(X, Y, validation_split = 0.1, epochs = 100000,
        callbacks = [earlystop, checkpoint])
```