Rekurrente Neuronale Netzwerke

Benjamin Roth

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilian-Universität München
beroth@cis.uni-muenchen.de

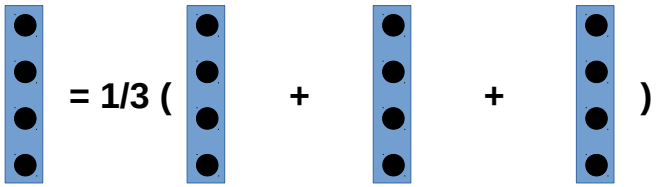
Rekurrente Neuronale Netzwerke: Motivation

Wie kann man ...

- ... am besten eine Sequenz von Wörtern als Vektor repräsentieren?
- ... die gelernten Wort-Vektoren effektive kombinieren?
- ... die für eine bestimmte Aufgabe relevante Information (bestimmte Merkmale bestimmter Wörter) behalten, unwesentliches unterdrücken?

Rekurrente Neuronale Netzwerke: Motivation

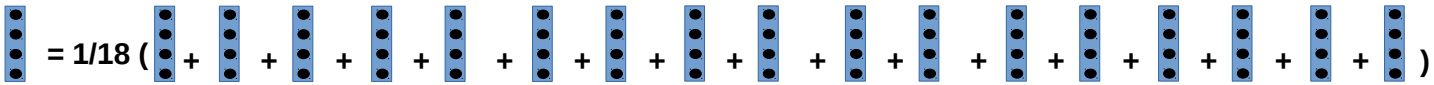
Bei kurzen Phrasen: Durchschnittsvektor evtl. Möglichkeit:



London Symphony Orchestra

$\Rightarrow$ employer?

Bei langen Phrasen problematisch.



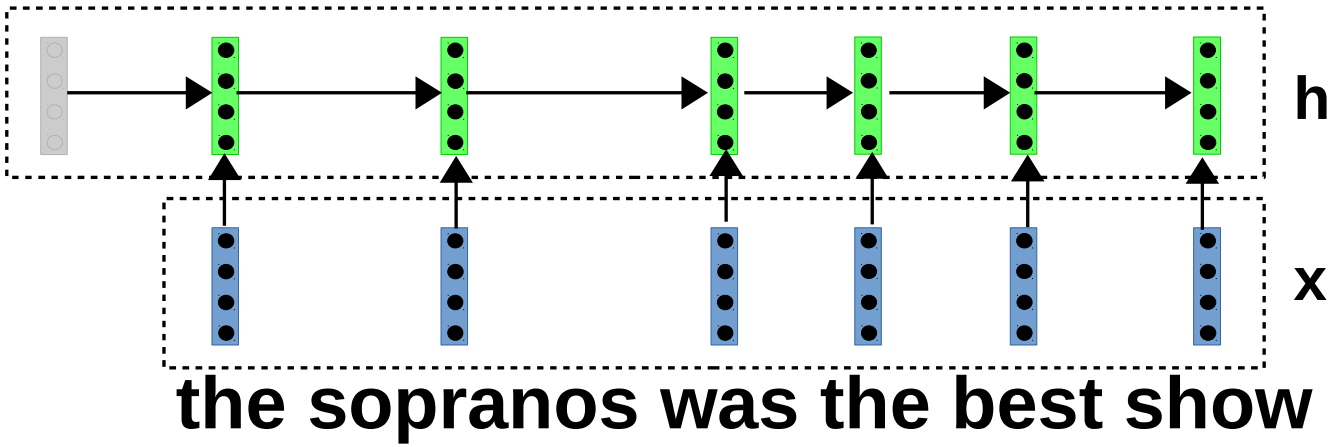
The sopranos was probably the last best show to air in the 90's. its sad that its over

- Reihenfolge geht verloren.
- Es gibt keine Parameter, die schon bei der Kombination zwischen wichtiger und unwichtiger Information unterscheiden können. (Erst der Klassifikator kann dies versuchen).

Rekurrente Neuronale Netzwerke: Idee

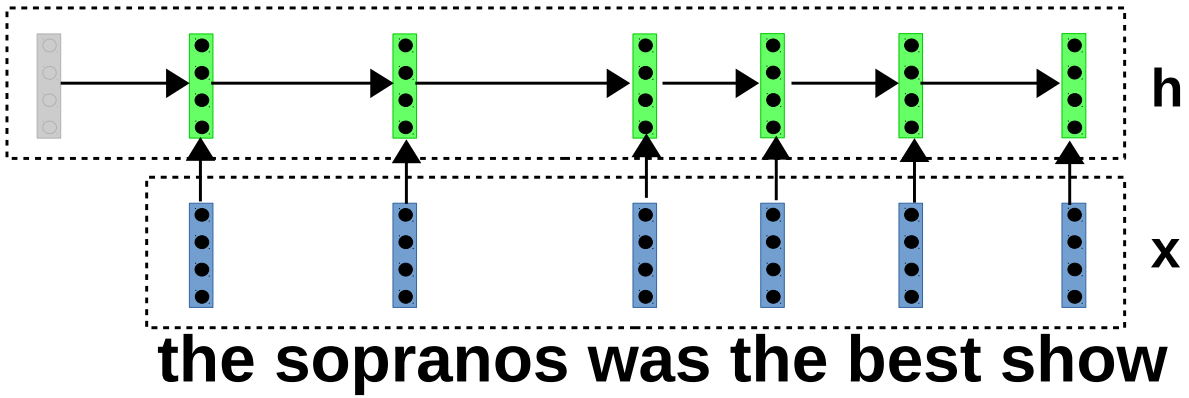
- Berechne für jede Position ("Zeitschritt", "time step") im Text eine Repräsentation, die alle wesentliche Information bis zu dieser Position zusammenfasst.
- Für eine Position t ist diese Reträsentation ein Vektor $\mathbf{h}^{(t)}$ (hidden representation)
- $\mathbf{h}^{(t)}$ wird rekursiv aus dem Wortvektor $\mathbf{x}^{(t)}$ und dem hidden Vektor der vorhergehenden Position berechnet:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$$



Rekurrente Neuronale Netzwerke

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$$



- Der hidden Vektor im letzten Zeitschritt $\mathbf{h}^{(n)}$ kann dann zur Klassifikation verwendet werden (*"Sentiment des Satzes?"*)
- Als Vorgänger-Repäsentation des ersten Zeitschritts wird der $\mathbf{0}$-Vektor verwendet.

Rekursive Funktion f

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$$

- Die Funktion f nimmt zwei Vektoren als Eingabe und gibt einen Vektor aus.
- Die Funktion f ist in den meisten Fällen eine Kombination aus:
  - **Vektor-Matrix-Multiplikation**
  - und einer **nicht-linearen Funktion** (z.B. logistic Sigmoid), die auf alle Komponenten des Ergebnisvektors angewendet wird.

$$\mathbf{h}^{(t)} = \sigma(\mathbf{W}[\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}])$$

Meist wird noch ein Bias-Vektor **b** hinzu addiert, den wir zu Vereinfachung weglassen.

Rekursive Funktion f

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) = \sigma(\mathbf{W}[\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}])$$

- **Vektor-Matrix-Multiplikation**:
  - Einfachste Form einen Vektor auf einen Vektor abzubilden.
  - Zunächst werden die Vektoren $\mathbf{h}^{(t-1)}$ (k Komponenten) und $\mathbf{x}^{(t)}$ (m Komponenten) aneinander gehängt (konkateniert):
    - Ergebnis $[\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}]$ hat Größe $k + m$ (Anzahl Vektor-Komponenten).
  - Gewichtsmatrix W (Größe: $k \times (k + m)$)
    - dieselbe Matrix für alle Zeitschritte (*weight sharing*)
    - wird beim Trainieren des RNN optimiert.

Rekursive Funktion f

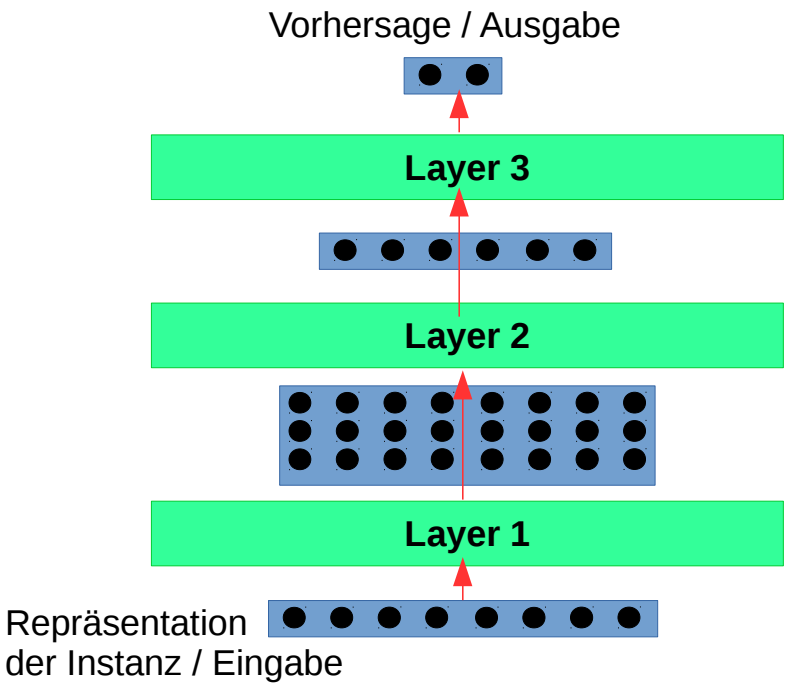
$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) = \sigma(\mathbf{W}[\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}])$$

- **Nicht-linearen Funktion**
  - Beispiele: Sigmoid, Tanh (=skalierter Sigmoid, zwischen $-1 \ldots 1$), Softmax, ReLu ($=max(0, x)$)
  - Wird auf alle Komponenten des Ergebnisvektors angewendet.
  - Notwendig, damit das Netzwerk qualitativ etwas anderes als eine Lineare Kombination der Eingaben ($\sim$ Durchschnittsvektor) berechnen kann.

Neuronale Netzwerke: Terminologie

Layers

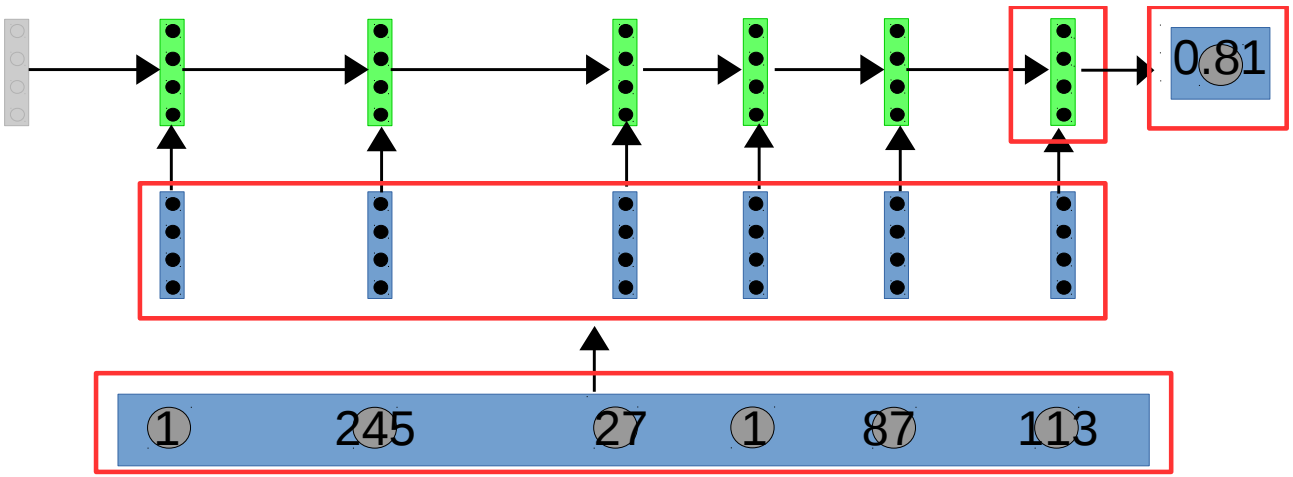
- Konzeptionell beschreibt man ein neuronales Netz als aus mehreren Schichten (*Layers*) aufgebaut.
- Jede Layer nimmt einen Vektor (oder eine Matrix) als Eingabe, und gibt einen Vektor (oder eine Matrix) aus.
- Die Größe der Ausgabe muss nicht mit der Größe der Eingabe übereinstimmen (auch Vektor $\leftrightarrow$ Matrix möglich).
- Die Ausgabe der vorhergehenden Layer ist die Eingabe für die nächste Layer.



Welche Layers gibt es in unserem Beispiel (Vorhersage von Sentiment mit RNN)?

Layers bei Vorhersage von Sentiment mit (einfachem) RNN

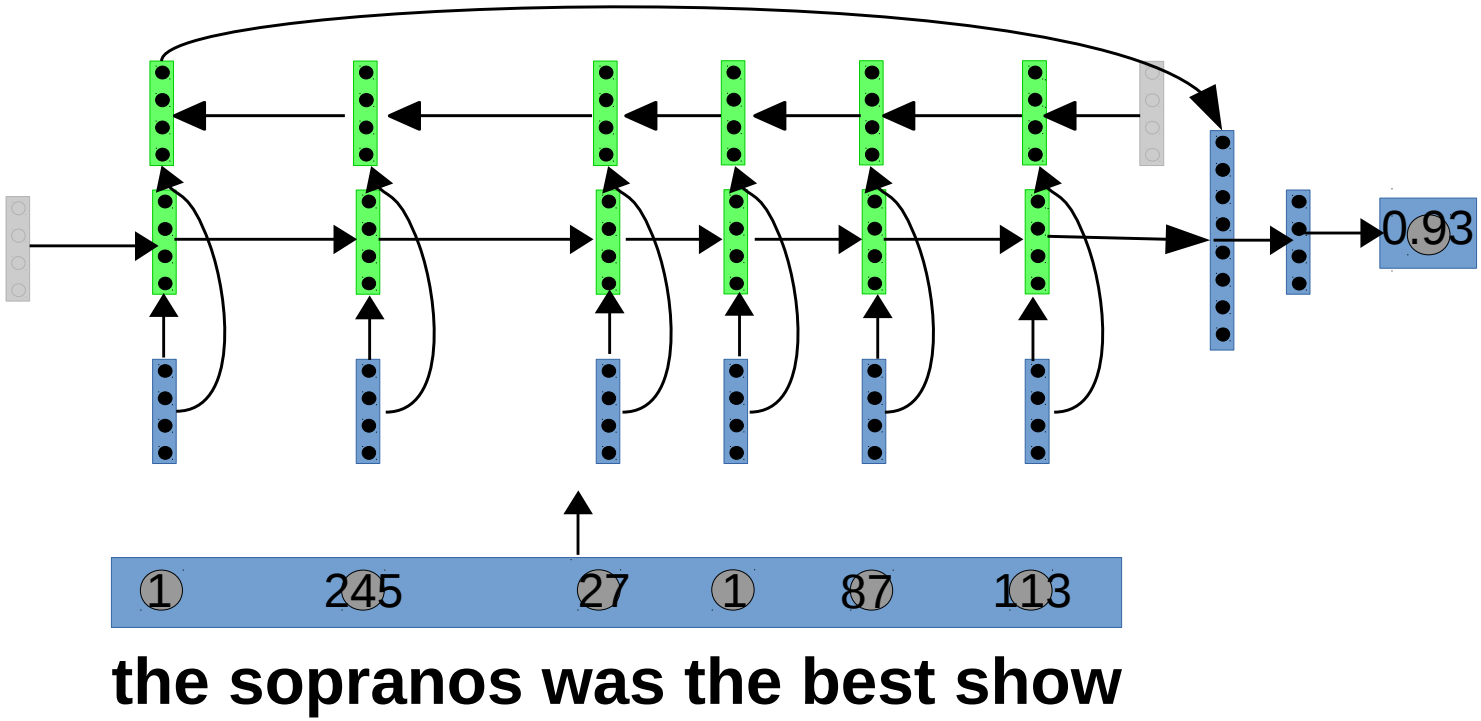
- Eingabe: Vektor mit Word-ids
- Layer 1 (Embedding): Lookup von Wort-Vektoren für ids (Vektor→Matrix)
- Layer 2 (RNN): Berechnung des Satz-Vektors aus Wort-Vektoren (Matrix→Vektor)
- Layer 3: Berechnung der Wahrscheinlichkeit für positives Sentiment aus Satz-Vektor
  (Vektor→Reelle Zahl, dargestellt als Vektor mit 1 Element)



Rot umrandet: Eingaben/Ausgaben

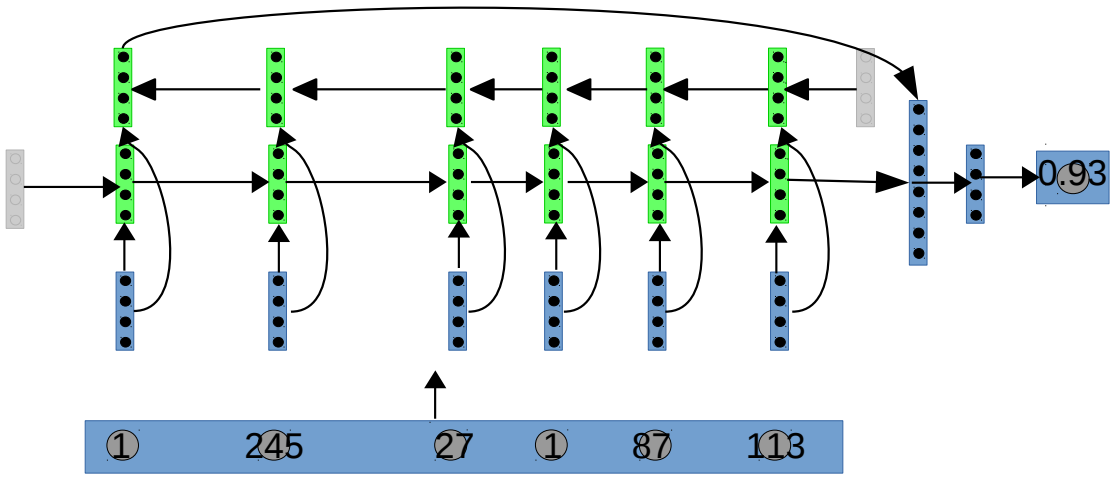
Vorhersage mit RNN: Mögliche Erweiterungen (1)

- Ein zweites RNN kann den Satz von rechts nach links verarbeiten: Die beiden RNN-Repräsentationen werden dann konkateniert.



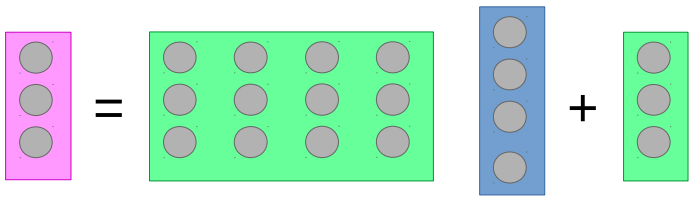
Vorhersage mit RNN: Mögliche Erweiterungen (2)

- Vor der Vorhersage können mehrere *Dense*-Layers hintereinandergeschaltet werden.
  - Eine Dense-Layer (auch: *fully connected layer*) entspricht einer Matrix-Multiplikation und Anwendung einer Nicht-Linearität
  - Eine Dense-Layer "übersetzt" Vektoren und kombiniert Information aus der vorhergehenden Layer.
  - Auch die Vorhersage-Layer ist meist eine Dense-Layer. (im Beispiel: übersetzung in einen Vektor der Größe 1; Nichtlinearität ist die Sigmoid Funktion)

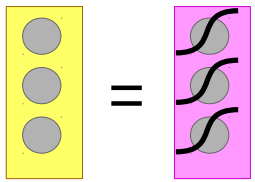


Dense-Layer: Veranschaulichung

- $\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$
  - $\mathbf{W}$ und $\mathbf{b}$ sind vom Modell zu lernende Parameter
  - Die Nichtlinearität σ wird elementweise angewendet
- $\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$



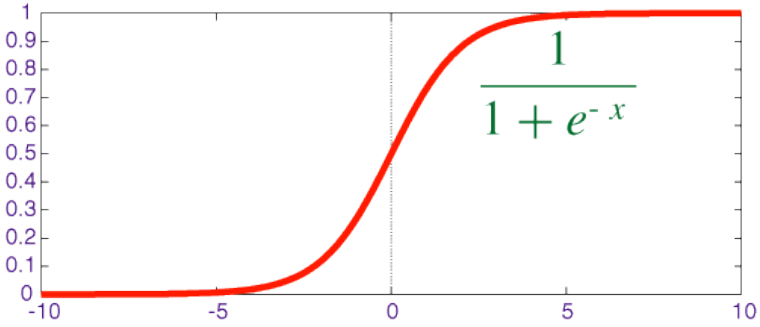
- $\mathbf{y} = \sigma(\hat{\mathbf{y}})$



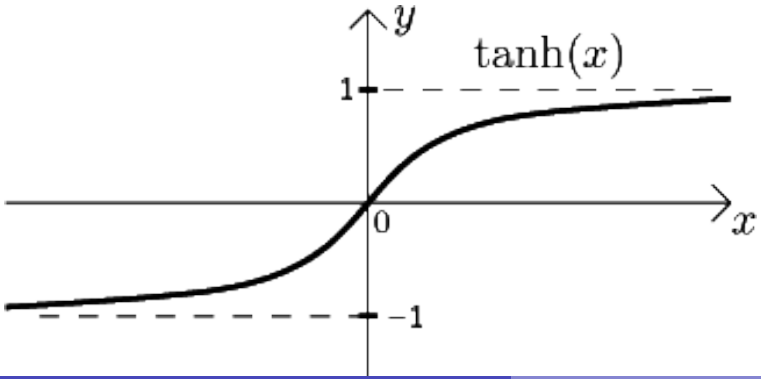
Hinweis: In einem einfachen RNN entspricht die rekursive Funktion einer Dense-Layer!

Häufig verwendete Nichtlinearitäten

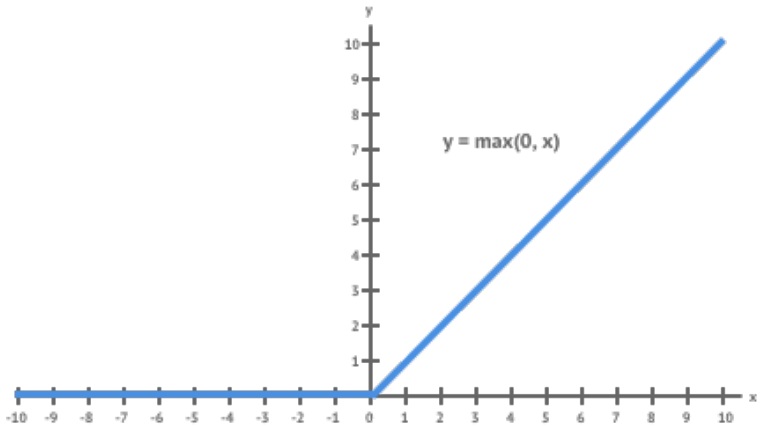
- Logistic Sigmoid: $\mathbf{y_i} = \sigma(\mathbf{x_i})$
  Wertebereich zwischen $0 \ldots 1$, kann als WK interpretiert werden.



- Tanh:
  $\mathbf{y_i} = \tanh(\mathbf{x_i}) = 2\sigma(2\mathbf{x_i}) - 1$
  Wie Logistic Sigmoid, aber Wertebereich zwischen $-1 \ldots 1$



- ReLu: $\mathbf{y_i} = \max(0, \mathbf{x_i})$



- Softmax:
  - $$\mathbf{y_i} = \frac{e^{(x_i)}}{\sum_j e^{(x_j)}}$$
  - Normiert die Ausgabe der vorangehenden Layer zu einer Warscheinlichkeitsverteilung
  - Meist für Vorhersage-Layer verwendet (WK für Ausgabe-Klassen)

Hinweis zum Lernen der Modellparameter

- Ein Neuronales Netzwerk ist eine aus einfachen Einheiten aufgebaute Funktion, mit einem Vektor als Eingabe (z.B. Word-Ids eines Satzes), und einem Vektor als Ausgabe (z.B. WK des positiven Sentiment).
- Für einen Datensatz kann nun eine Kostenfunktion berechnet werden, z.B. die negative Loglikelihood.
  - (negative log-) Wahrscheinlichkeit, die das Modell den Labels der Trainingsdaten zuweist.
  - Manchmal wird auch die Bezeichnung **Cross-Entropy** verwendet
- Die Parameter können dann (ähnlich wie bei Word2Vec) mit Stochastic Gradient Descent optimiert werden.
  - Parameter sind z.B. Wort-Embeddings, Gewichtmatrizen der Dense-Layers, ... etc.
  - Anders als bei Word2Vec wird bei NN ein Parameter-Update meist für eine *Mini-Batch* von 10-500 Trainingsinstanzen durchgeführt.
  - Es stehen mehrere Erweiterungen von SGD zur Verfügung (RMS-Prop, Adagrad, Adam, ...)

Neuronale Netzwerke: Implementierung mit Keras

Introduction

What is Keras?

- Neural Network library written in Python
- Designed to be minimalistic & straight forward yet extensive
- Built on top of either Theano or TensorFlow

Keras strong points:

- Simple to get started, powerful enough to build serious models
- Takes a lot of work away from you.
- Reasonable defaults (e.g. weight matrix initialization).
- Little redundancy. Architectural details are inferred when possible (e.g. input dimensions of intermediate layers).
- highly modular; easy to expand

Keras: Idea

```
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 entprechende Objekt erzeugt, und dieses zum Modell hinzugefügt.
  - Die jeweiligen Layers übernehmen die Ausgabe der vorhergehenden Layers als ihre Eingabe.

Keras: Idea

```
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 spezifiert, 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` für 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.
- 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.

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

?

- Vorteile / Nachteile wenn man vortrainierte Wortvektoren benutzt, und diese nicht weiter optimiert?

?

- Vorteile / Nachteile wenn man vortrainierte Wortvektoren benutzt, und diese nicht weiter optimiert?
- *Vorteil: Für einen bestimmten Task, wie z.B. Sentiment-Analyse hat man oft vergleichsweise wenige Trainingsdaten. Wortvektoren kann man unüberwacht auf großen Korpora trainieren, diese haben deshalb eine* **bessere Abdeckung**. *Außerdem hat das Modell weniger Parameter zu optimieren, weshalb* **weniger Gefahr des Overfitting** *besteht.*
- *Nachteil: Die verwendeten Wortvektoren passen u.U. nicht so gut zum Task, die relevanten Eigenschaften wurden beim unüberwachten Lernen der Vektoren nicht berücksichtigt* $\Rightarrow$ **Underfitting**
- *Hinweis: Ein guter Mittelweg ist oft, die Vektoren mit vortrainierten Vektoren zu initialisieren, und diese trotzdem noch weiter Task-spezifisch zu optimieren.*

Keras: RNN Layer

- Die zuvor vorgestellte Variante des RNN ist zwar ein mächtiges Modell, die Parameter sind aber 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)

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

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

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

- 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?

Keras: RNN Layer

- Anstelle des Endvektors kann auch eine Matrix ausgegeben werden, die für jede Position den Zustandsvektor **h** enthält: **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 vorhergender Layer.
  - Nichtlinearität ist ReLu oder Tanh.

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

- Als Ausgabe-Layer
  - Wahrscheinlichkeit einer Ausgabe.
  - Nichtlinearität ist Sigmoid (2 mögliche Klassen) oder Softmax (beliebig viele Klassen).

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

Training

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

- 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`

Training

`model.fit(...)`
Weitere Argumente:

- 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.

Vorhersage und Evaluierung

- `y_predicted = model.predict(x_dev)`
- `score, acc, ... = model.evaluate(x_dev, y_dev)`
  Gibt den Wert der Zielfunktion und der Metriken zurück (`loss` bzw. `metrics` von `model.compile`)

Hinweise

- Um mit Keras produktiv arbeiten zu können ist es wichtig, sich mit der API/Dukumentation vertraut zu machen!
- `https://keras.io/getting-started/sequential-model-guide/`
- 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. (Zu lange Listen werden abgeschnitten, zu kurze mit 0-Werten aufgefüllt) [1]

[1] Modul keras.preprocessing.sequence

Convolutional Neural Networks

- CNN können ähnlich einfach verwendet werden wie RNN's.
- Um z.B. für Sentiment Vorhersage ein CNN mit 50 Filtern (Ausgabedimensionen) und Filterweite 3 Wörtern zu erzeugen ...
- ... muss statt der Zeile model.add(LSTM(...)) ein CNN mit max-Pooling verwendet werden:

```
...
model.add(Convolution1D(nb_filter=50, filter_length=3, \
                        activation='relu', \
                        border_mode='same'))
model.add(GlobalMaxPooling1D())
...
```

Zusammenfassung

- RNNs: Erzeugt eine Sequenz von Vektoren (*hidden states*).
- Jeder hidden Vektor wird rekursiv aus dem vorhergehenden Vektor, und dem Wort-Embedding der aktuellen Position berechnet.
- Eine Sequenz kann z.B. durch den letzten hidden Vektor dargestellt werden.