Recurrent Neural Networks

Benjamin Roth

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

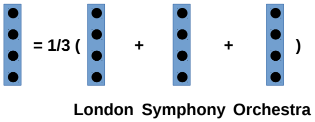
Recursive Neural Networks: Motivation

How do you ...

- ... best represent a sequence of words as a vector?
- ... combine the learned word vectors effectively?
- ... retain the information relevant to a particular task (certain features of particular words), suppress unessential aspects?

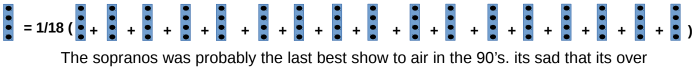
Recursive Neural Networks: Motivation

For short phrases: average vector could be one possibility



$\Rightarrow$ employer?

For long phrases problematic.

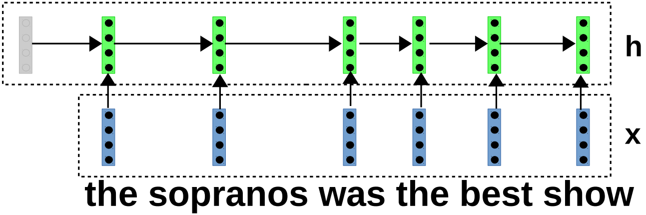


- Any information about the order of words is lost.
- There are no parameters that can already during combination distinguish between important and unimportant information. (Only the classifier can try this).

Recursive Neural Networks: Idea

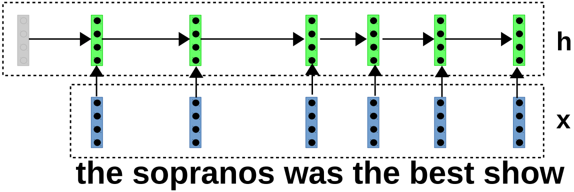
- Calculate for each position ("time step") in the text a representation that summarizes all essential information up to this position.
- For a position t this representation is a vector $\boldsymbol{h}^{(t)}$ (hidden representation)
- $\boldsymbol{h}^{(t)}$ is calculated recursively from the word vector $\boldsymbol{x}^{(t)}$ and the hidden vector of the previous position:

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



Recursive Neural Networks

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



- The hidden vector in the last time step $\boldsymbol{h}^{(n)}$ can then be used for classification (*"Sentiment of the sentence?"*)
- The predecessor representation of the first time step uses the $\mathbf{0}$ vector (containing only zeros).

Recursive function f

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

- The f function takes two vectors as input and outputs a vector.
- The function f is in most cases a combination of:
  - **Vector matrix multiplication**:
  - and a **non-linear function** (e.g., logistic sigmoid) applied to all components of the resulting vector.

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

Usually a bias vector $\boldsymbol{b}$ is added, which is sometimes omitted for simplicity.

Recursive function f

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

- **Vector matrix multiplication**:
  - Simplest form of mapping a vector onto a vector.
  - First, the vectors $\boldsymbol{h}^{(t-1)}$ (k components) and $\boldsymbol{x}^{(t)}$ (m components) are concatenated:
    - Result $[\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}]$ has $k + m$ components.
  - Weight matrix W (size: $k \times (k + m)$)
    - the same matrix for all time steps (*weight sharing*)
    - is optimized when training the RNN.

Recursive function f

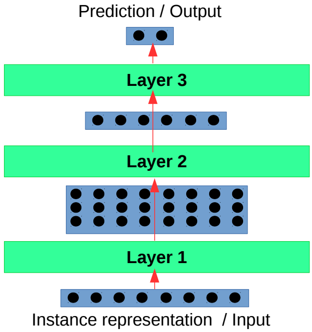
$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}) = \sigma(\boldsymbol{W}[\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}] + \boldsymbol{b})$$

- **Non-linear function**
  - Examples: Sigmoid, Tanh (= scaled sigmoid, between $-1 \dots 1$), Softmax, ReLu ($= max(0, x)$)
  - Applied to all components of the resulting vector.
  - Necessary so that the network can compute interesting, non-linear interactions, such as the effect of negation.

Neural Networks: Terminology

Layers

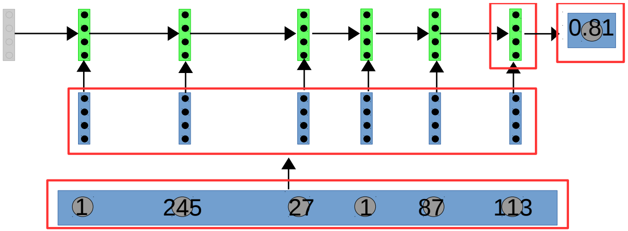
- Conceptually, a neural network is composed of several (*layers*).
- Each layer is a function that takes a vector (or matrix) as the input, and outputs a vector (or matrix).
- The size of the output does not have to match the size of the input (also vector $\leftrightarrow$ matrix possible).
- The output of the previous layer is the input for the next layer.



Which layers are there in our example (prediction of sentiment with RNN)?

Layers predicting sentiment with (simple) RNN

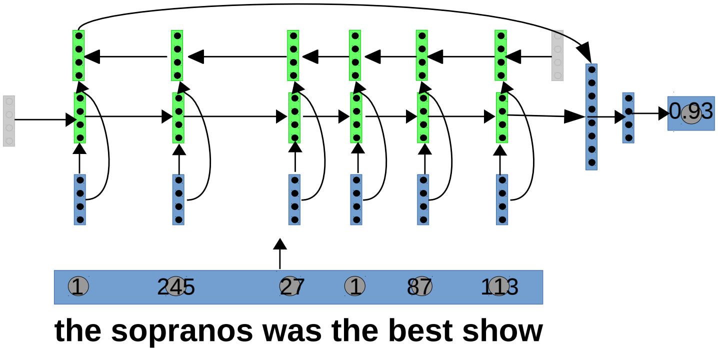
- Input: vector with word-ids
- Layer 1 (Embedding): Lookup of word vectors for ids (vector→matrix)
- Layer 2 (RNN): Calculation of the sentence vector from word vectors (matrix→vector)
- Layer 3: Calculation of the probability for positive sentiment from the sentence vector
  (vector→Real number, represented as a vector with 1 element)



Outlined in red: inputs / outputs

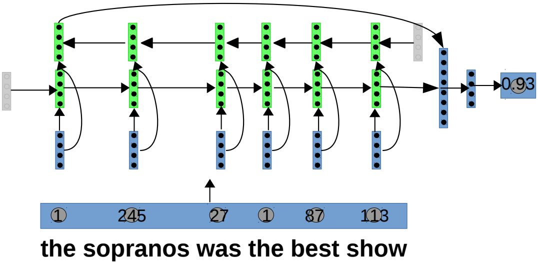
Prediction with RNN: Possible extensions (1)

- A second RNN can process the sentence from right to left: The two RNN representations are then concatenated.



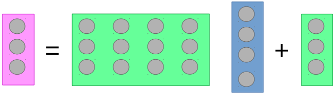
Prediction with RNN: Possible extensions (2)

- Before the prediction, several *Dense* layers can be cascaded.
  - A dense layer (also: *fully connected layer*) corresponds to a matrix multiplication (+ bias) and application of a non-linearity
  - A Dense layer " translates " vectors and combines information from the previous layer.
  - Usually, the prediction layer is a dense layer. (in the example: translation into a vector of size 1, nonlinearity is the sigmoid function)

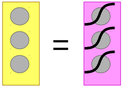


Dense-Layer: illustration

- $\boldsymbol{y} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$
  - $\boldsymbol{W}$ uand $\boldsymbol{b}$ are parameters that have to be learned by the model
  - The nonlinearity σ is applied element by element
- $\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$



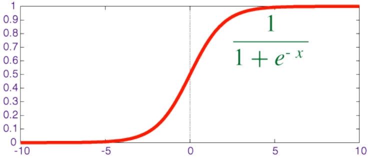
- $\boldsymbol{y} = \sigma(\hat{\boldsymbol{y}})$



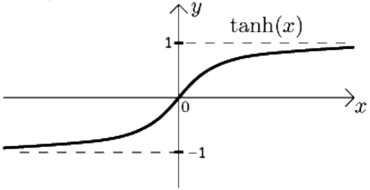
Note: In a simple RNN, the recursive function corresponds to a dense layer!

Frequently used nonlinearities

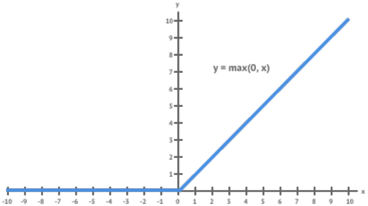
- Logistic Sigmoid: $\boldsymbol{y_i} = \sigma(\boldsymbol{x_i})$
  Value range between $0 \dots 1$, can be interpreted as a **probability**.



- Tanh:
  $\boldsymbol{y_i} = \tanh(\boldsymbol{x_i}) = 2\sigma(2\boldsymbol{x_i}) - 1$
  Like Logistic Sigmoid, but value range between $-1 \dots 1$



- ReLU (*rectified linear unit*):
  $\boldsymbol{y_i} = \max(0, \boldsymbol{x_i})$



- Softmax:
  - $$\boldsymbol{y_i} = \frac{e^{(x_i)}}{\sum_j e^{(x_j)}}$$
  - Normalizes the output of the preceding layers to a **probability distribution**
  - Mostly used in output layer for prediction

Note on learning the model parameters

- A neural network is a function built from simple units, with one vector as the input (e.g., word ids of a sentence), and another vector as the output (e.g., probability for positive sentiment).
- For a data set, a cost function can now be calculated, e.g. the negative log likelihood:
  - (negative log) probability that the model assigns to the annotated labels of the data set.
  - Sometimes also called **cross-entropy**.
- The parameters can then be optimized (similar to Word2Vec) with Stochastic Gradient Descent.
  - Parameters are e.g. Word Embeddings, Weight Layers of Dense Layers, ... etc.
  - Unlike Word2Vec, NN usually performs a parameter update on a *mini-batch* of 10-500 training instances.
  - Several extensions of SGD are available (RMS-Prop, Adagrad, Adam, ...)

Neural Networks: Implementation with Keras

Introduction

What is Keras?

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

Keras strong points:

- Easy 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, masking).
- 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()` creates a model in which layers can be sequentially stacked on each other.
  - For each layer, the corresponding object is first created and added to the model.
  - The added layer take over the output of the previous layer as its input.

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`: When the specification of the model is completed, it can be compiled:
  - It is specified which learning algorithm should be used.
  - Which cost function should be minimized.
  - And what additional metrics should be calculated for evaluation.
- `model.fit`: Training (adjust the parameters in all layers)

Keras: Embedding Layer

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

- Provides word vectors of size `output_dim` for a vocabulary of size `input_dim`.
  - Often the first layer in a model.
  - Input per instance: vector with word id's
  - Output per instance: matrix; sequence of word vectors.
- The parameters (word vectors) of the embedding layer
  - ... can be initialized with pre-trained vectors (Word2Vec), or at random.
  - ... if you use pre-trained word vectors, further optimization of the word vectors is sometimes not necessary.

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

?

- Advantages / disadvantages of using pre-trained word vectors and not optimizing them further?

?

- Advantages / disadvantages of using pre-trained word vectors and not optimizing them further?
- *Advantage: For a specific task, such as Sentiment analysis, often comparatively little training data is available. Word vectors can be trained unsupervised on large corpora, these therefore have a* **better coverage***. In addition, the model has fewer parameters to optimize, which is why* **there is less risk of overfitting***.*
- *Disadvantage: The word vectors used may not fit the task, the relevant properties were not taken into account in the unsupervised learning of the vectors* $\Rightarrow$ **Underfitting**
- *Note: A good middle ground is often to initialize the vectors with pre-trained vectors, and still further optimize them on the task-specific training data.*

Keras: RNN Layer

- Although the previously introduced variant of the RNN is an expressive model, the parameters are difficult to optimize (*vanishing gradient problem*).
- Extensions of the RNN, which facilitate the optimization of the parameters, are e.g. **LSTM** (long short-term memory network) and **GRU** (gated recurrent unit network)

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

- Two RNNs (left-to-right and right-to-left). output are the concatenated end vectors (as in the example above):

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

- Instead of the end vector, a matrix can also be output which contains the state vector $\boldsymbol{h}$ for each position:

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

For which computer linguistic tasks is it necessary to have access to the state vector at each position?

Keras: RNN Layer

- Instead of the end vector, a matrix can also be output which contains the state vector $\boldsymbol{h}$ for each position: **For which computer linguistic tasks is it necessary to have access to the state vector at each position?**

Whenever a prediction needs to be made for each position, e.g. part of speech tagging.

Keras: Dense Layer

Two options:

- As an intermediate layer
  - Combines information from previous layers.
  - Nonlinearity is ReLu or Tanh.

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

- As output layer
  - Probability of an output.
  - Non-linearity is sigmoid (probability of output 1-vs-0) or softmax (any number of classes, one-hot-encoding).

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

Training

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

- Loss functions:
  - `binary_crossentropy` if only one class is predicted (sigmoid activation)
  - `categorical_crossentropy` if probability distribution over several classes (Softmax activation)
- Optimizer: `adam`, `rmsprop`, `sgd`

Training

`model.fit(...)`

Other arguments:

- Hyper-parameters
  - `batch_size`: how many instances should be used for one optimization step. (Optimization step $\neq$ training iteration)
  - `epochs`: How many training iterations should be performed.
  - ...
- `validation_data`: Tuple `(features_dev, labels_dev)`
  Development data, e.g. to monitor training progress.

Prediction and evaluation

- `y_predicted = model.predict(x_dev)`
- `score, acc, ... = model.evaluate(x_dev, y_dev)`
  Returns the value of the objective function and the metrics (`loss` or `metrics` of `model.compile`)

Hints

- In order to be productive with Keras, it is important to become familiar with the API / Documentation!
- `https://keras.io/getting-started/sequential-model-guide/`
- Keras expects `inputs` as numpy arrays. Lists of various lengths (e.g., sentence representations) can be converted to a numpy array of a given number of columns by the `pad_sequences(list_of_lists, max_length)` command. (Too long lists are truncated, shorter ones are filled with 0 values) [1]

[1] Modul keras.preprocessing.sequence

Convolutional Neural Networks

- CNNs can be used just as easily as RNNs.
- For example, to generate a CNN with 50 filters (output dimensions) and filter width 3 words for sentiment prediction ...
- ... instead of the line model.add (LSTM (...)), a CNN with max pooling must be used:

```
...
model.add(Conv1D(filters=50, kernel_size=3, \
          activation='relu', padding='same'))
model.add(GlobalMaxPooling1D())
...
```

Summary

- RNNs: Creates a sequence of vectors (*hidden states*).
- Each hidden vector is calculated recursively from the previous vector, and the word-embedding of the current position.
- A sequence may e.g. represented by the last hidden vector.