Lecture 13:

Conclusion (part 1)

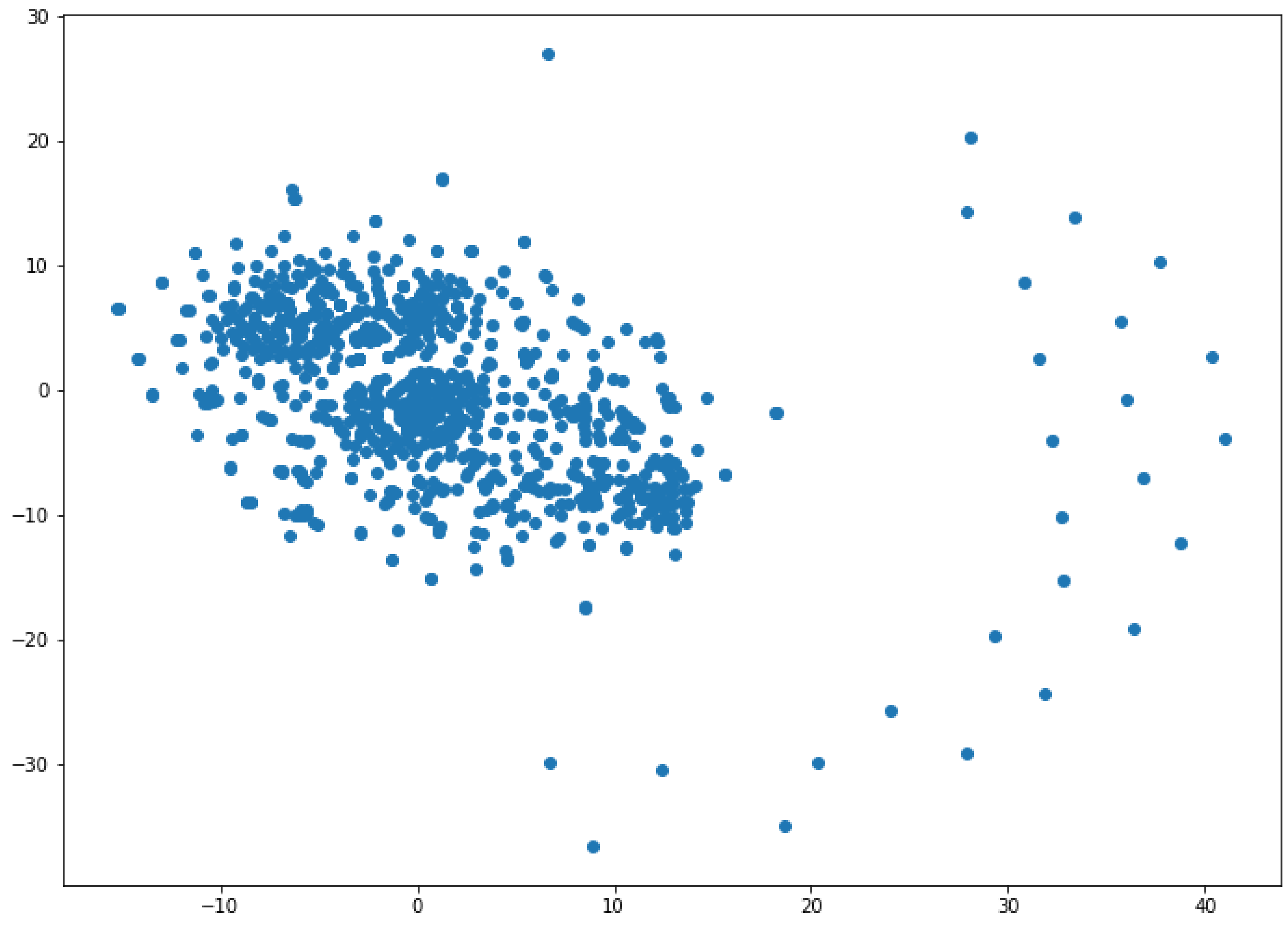
Word2vec in Python

- Visualization like t-SNE can be used to inspect the pretrained word2vec resulting vectors

```
# Collect pretrained word2vec vectors of all words in the
# 20newsgroups vocabulary (use zeros for words we do not find)
word2vec_allvectors=numpy.zeros((len(remainingvocabulary),300))
for i in range(len(remainingvocabulary)):
    try:
        tempvector=word2vec_pretrainedvectors.wv[remainingvocabulary[i]]
        word2vec_allvectors[i,:]=tempvector
    except:
        continue
# Take a random subset of 1000 words
wordsubsetindices=numpy.random.permutation(len(remainingvocabulary))
# Create and plot a 2D t-SNE visualization
import sklearn.manifold
tsnemodel = sklearn.manifold.TSNE(n_components=2, verbose=1,
perplexity=20, n_iter=400)
tsneplot3 = tsnemodel.fit_transform(\
    word2vec_allvectors[wordsubsetindices[0:1000],:])
myfigure, myaxes = matplotlib.pyplot.subplots();
myaxes.scatter(tsneplot3[:,0],tsneplot3[:,1]);
```

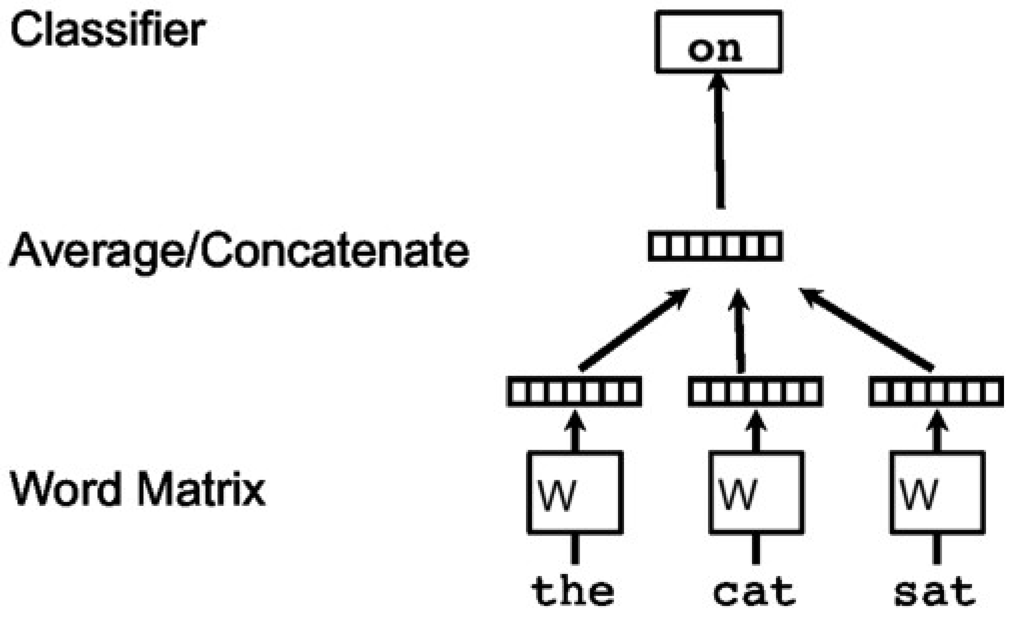
Word2vec in Python

- Result seems to show several subgroups of words with close-by meanings, plus outliers:



Paragraph vector models

- Quoc Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. International Conference on Machine Learning, 2014. ArXiV: https://arxiv.org/pdf/1405.4053.pdf
- Reminder: the continuous bag of words (CBOW) model predicted a central word based on context words near it.



CBOW model (picture from Le & Mikolov 2014). Here the context is three previous words "the cat sat" and the central word is the next word (here "on").

$$p(w|\boldsymbol{C}(w);\boldsymbol{\theta})=\frac{e^{\text{affinity}(w,\boldsymbol{C}(w))}}{\sum_{v=1}^{V} e^{\text{affinity}(v,\boldsymbol{C}(w))}}$$

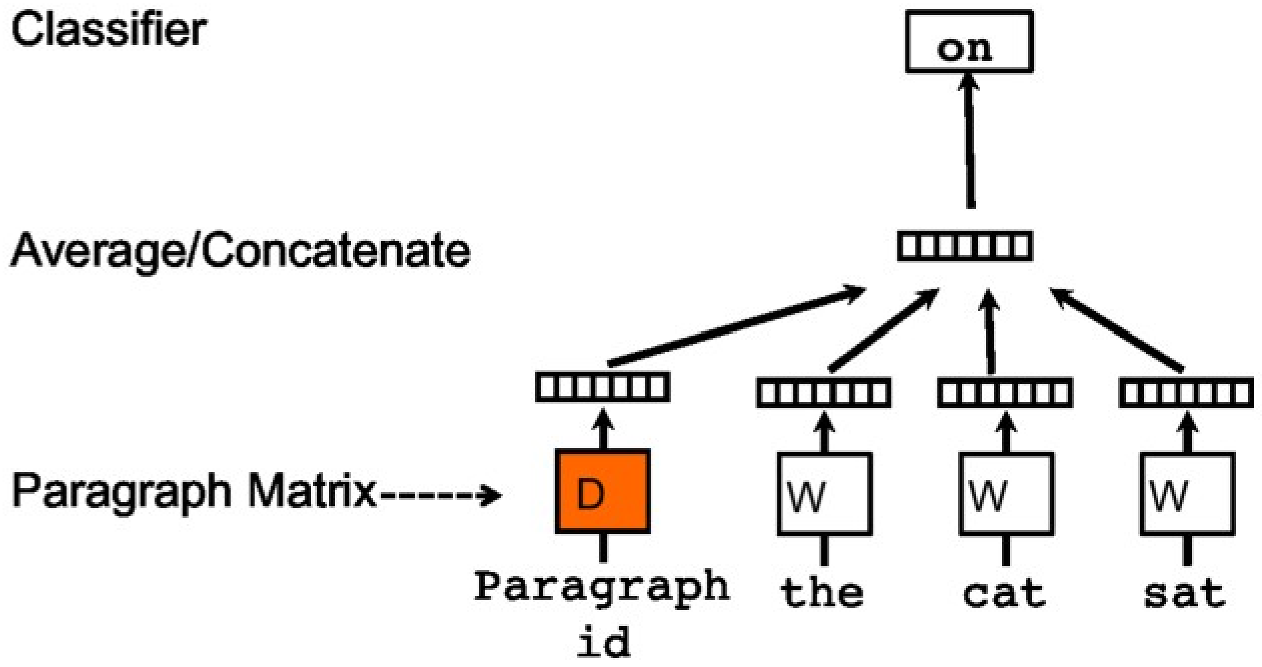
$$\text{affinity}(w,\boldsymbol{C}(w))=\boldsymbol{\phi}_w^T\left(\frac{1}{2R}\sum_{c\in\boldsymbol{C}(w)}\boldsymbol{\psi}_c\right)+\phi_{0,w}$$

Usually includes also a bias term

Paragraph vector models

- **Paragraph vector model** (also called "Distributed Memory Model of Paragraph Vectors, PV-DM"):
  - in addition to the context words, the central word is predicted also based on a paragraph id number.
  - Context words are mapped to vectors like before
  - Every paragraph is mapped to a unique **paragraph vector**.
  - The paragraph vector is shared for all contexts in the same paragraph, but is different for different paragraphs
- The context-word vectors and the paragraph vector are either **averaged** or **concatenated** to form the context representation.
- In the concatenation version, the paragraph vectors can have different dimensionality from the context-word vectors

Paragraph vector models



Paragraph vector model (picture from Le & Mikolov 2014). Here the context is "the cat sat" + the paragraph id and the word to be predicted is the next word (here "on")

- **Equations:**

Parameters are found by maximizing log-likelihood

$$p(w|\boldsymbol{C}(w);\boldsymbol{\theta})=\frac{e^{\text{affinity}(w,\boldsymbol{C}(w))}}{\sum_{v=1}^{V} e^{\text{affinity}(v,\boldsymbol{C}(w))}}$$

$$\text{affinity}(w,\boldsymbol{C}(w))=\boldsymbol{\phi}_w^T\left(\frac{1}{2R+1}\left(\sum_{c\in\boldsymbol{C}(w)}\boldsymbol{\psi}_c+\boldsymbol{\gamma}_{\text{Paragraph-id}(w)}\right)\right)+\phi_{0,w}$$ Averaging version

$$\text{affinity}(w,\boldsymbol{C}(w))=\boldsymbol{\phi}_w^T[\boldsymbol{\psi}_{c_1},\ldots,\boldsymbol{\psi}_{c_{|\boldsymbol{C}(w)|}},\boldsymbol{\gamma}_{\text{Paragraph-id}(w)}]+\phi_{0,w}$$

Concatenation version. Here $\boldsymbol{\phi}_w$ has a much higher dimensionality than in the averaging version!

Paragraph vector models

- Parameters are found by maximizing the likelihood (probability of the central words given the contexts):

$$\prod_{s=1}^{M}\prod_{i=1}^{N^{(s)}} p\left(w_i^{(s)} \middle| \boldsymbol{C}\left(w_i^{(s)}\right);\boldsymbol{\theta}\right)$$

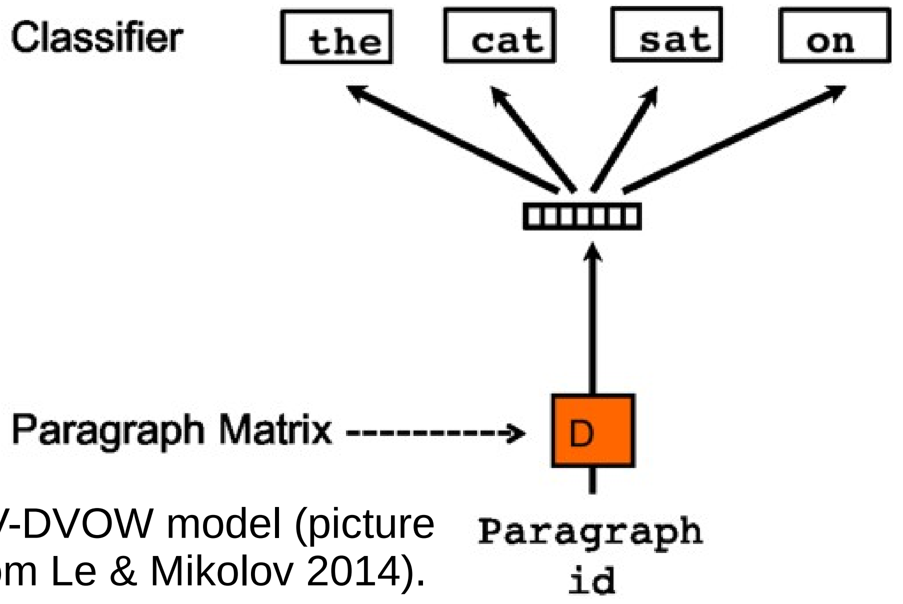
  with respect to the word vectors, context-word vectors, and paragraph vectors.

- In a new prediction situation (predicting the next word in a new paragraph), the paragraph vector would not be available yet. It must be optimized first:
  - Word vectors and context-word vectors are kept at their previously optimized values.
  - Maximize the probability of the known words in the new paragraph (given their contexts) with respect to the paragraph vector.
  - Then use the paragraph vector to predict the new word

- In this model the context-word vectors $\boldsymbol{\psi}_c$ are used as the embedding of words, since the word vectors $\boldsymbol{\phi}_w$ are high-dimensional

Paragraph vector models

- In the concatenation version, the order of concatenating the context-words affects the result. Taking word order into account can be good.
- Le and Mikolov (2014) suggest a "Distributed Bag of Words version of Paragraph Vector (PV-DBOW)" that does not consider word order
- It uses only the paragraph vector to predict randomly picked words from the paragraph. Similar to the skip-gram model, which used a central word to predict context words.
- For each paragraph, 1) sample a random text window, and a random context-word word from the window. 2) Predict the context-word given the paragraph vector. 3) Repeat several times.



PV-DVOW model (picture from Le & Mikolov 2014).

$$\prod_{i \in \text{paragraphs}} \prod_{c \in \text{samples}_i} p(c|i;\boldsymbol{\theta})$$ Likelihood

$$p(c|i;\boldsymbol{\theta}) = \frac{e^{\text{affinity}(c,i)}}{\sum_{u=1}^{U} e^{\text{affinity}(u,i)}}$$ probability

$$\text{affinity}(c,i) = \boldsymbol{\psi}_c^T \boldsymbol{\gamma}_i + \psi_{0,c}$$ affi-nity

Paragraph vector models

- Even though they are called "paragraph vectors", the idea applies to any blocks of words: for example, sentences.
- Example application: sentiment classification of sentences (positive or negative). Data: Stanford Sentiment Treebank Dataset, 11855 sentences from Rotten Tomatoes movie reviews.
- In training data, use sentences as "paragraphs", and learn paragraph vectors $\boldsymbol{\gamma}_i$ for them.
- Learn a logistic regression classifier to predict classes of training sentences:

$$p(\text{positive}|i) = \frac{1}{1+e^{-(w_i^T \boldsymbol{\gamma}_i + w_0)}}$$

Class probability

$$\sum_{i\in\text{positive}} \log p(\text{positive}|i) + \sum_{i\in\text{negative}} \log(1-p(\text{positive}|i))$$

Log-likelihood: optimize with respect to weights **w**

- For test sentences, first optimize a paragraph vector for them, then feed it to the logistic regression classifier to predict class probability
- Le and Mikolov use a concatenation of paragraph vectors from the two models: PV-DBOW and PV-DM
- Result: 12.2% classification error rate.

Paragraph vector models

- Example application 2: sentiment classification of documents (positive or negative). Data: IMDB movie reviews data set, 25000 train + 25000 labeled test reviews + 50000 unlabeled reviews. Probably the same as the one at https://ai.stanford.edu/~amaas/data/sentiment/
- In training data, use entire reviews as "paragraphs", and learn paragraph vectors $\boldsymbol{\gamma}_i$ for them.
- Learn a logistic regression classifier to predict classes of training reviews, like before.
- For test reviews, first optimize a paragraph vector for them, then feed it to the logistic regression classifier to predict class probability
- Result: 7.42% classification error rate.

Paragraph vector models

- Example application 3: information retrieval.
- Data: triplets of search result snippets: (s1, s2, s3). In each, s1 and s2 are snippets of webpages from the first page of search results for the same query; s3 is a snippet from another, randomly chosen query.
- Learn paragraph vectors for each snippet, then compute Euclidean distances between s1, s2, and s3.
- Count a retrieval success if s1 was closer to s2 than s3, error otherwise
- Result: 3.82% error rate
- Le and Mikolov (2014) compared this to:
  - bag-of-words (unclear what data representation and distance metric was used - e.g. Euclidean distance of TF-IDF vectors?). 8.10% error rate
  - bag-of-bigrams (unclear what data representation and distance metric was used - e.g. Euclidean distance of TF-IDF bigram vectors?). 7.28% error rate
  - bag-of-bigrams version that tried to learn a weighting matrix for keeping s1 closer to s2 than s3. 5.67% error rate
  - learning word vectors for words, and averaging the word vectors for each snippet. 10.25% error rate

Paragraph vector models

- Python's gensim library includes learning of paragraph vector models (there called "doc2vec")
- They use a "hierarchical softmax" to compute log-likelihood over the output vocabulary, or a negative sampling model (unclear what exactly that means for the paragraph vector case)

```
import gensim
# We need to create a tagged version of each document
gensim_tagged_docs=[]
for k in range(len(mycrawled_prunedtexts)):
    doctag='doc'+str(k)
    tagged_document= \
        gensim.models.doc2vec.TaggedDocument( \
        mycrawled_prunedtexts[k],[doctag])
    gensim_tagged_docs.append(tagged_document)
# Create a dictionary from the documents
gensim_dictionary = gensim.corpora.Dictionary(gensim_docs)
```

Paragraph vector models

- Python's gensim library includes learning of paragraph vector models (there called "doc2vec")
- They use a "hierarchical softmax" to compute log-likelihood over the output vocabulary, or a negative sampling model (unclear what exactly that means for the paragraph vector case)

```
# Train the word2vec model
# The dm_concat parameter controls whether to concatenate
# or average word vectors when learning the paragraph
# vector  (see slides 5 and 6).
doc2vecmodel =
gensim.models.doc2vec.Doc2Vec(gensim_tagged_docs, \
    vector_size=10, window=5, min_count=1, \
    workers=4, dm_concat=0)
doc2vecmodel['doc986']
```

```
Out[296]:
array([-0.33598384, -0.29383156, -0.3040801 , -0.11860801, -0.15689434,
        0.12553768,  0.02579052, -0.05977593,  0.17080739, -0.02232333],
      dtype=float32)
```

Neural language models

- Models like word2vec and paragraph vector models use continuous-valued vector representations internally, but combine them only linearly, except a final softmax nonlinearity to get output probabilities.
- Neural language models extend this to use nonlinearities in intermediate computations.
- Generally, neural language models:
  - use as input a string of words, each encoded as 1-of-V vectors (where the only 1 is at the vocabulary index of the word, other elements are zero).
  - use a softmax equation at the output to compute probabilities that sum to 1 over several possibilities.
  - optimize parameters to maximize the likelihood (probability of observations)
- The output is computed by affinities to a feature **f** extracted from inputs, where **f** is computed in different ways by different neural models

$$p(w|\text{inputs};\boldsymbol{\theta})=\frac{e^{\boldsymbol{\phi}_w^T \boldsymbol{f}(\text{inputs})+\phi_{w,0}}}{\sum_{v=1}^{V} e^{\phi_v^T \boldsymbol{f}(\text{inputs})+\phi_{v,0}}}$$

weights $\boldsymbol{\phi}_w, \phi_{w,0}$ are network parameters

Neural language models

- In principle, **f** could be computed in any way from the inputs
- The basic idea in neural networks is to compute **f** as a composition of many functions: **f** = **f**5(**f**4(**f**3(**f**2(**f**1(inputs))))), so that each of the individual functions has a simple form and tunable parameters
- In a feedforward network, each **fi** is a layer of the network, and its K-dimensional output is computed by K neurons:

$$\boldsymbol{f}_i(\boldsymbol{x}) = [neuron_{i1}(\boldsymbol{x}), \ldots, neuron_{iK}(\boldsymbol{x})]$$

- A typical neuron computes a weighted linear sum of inputs, and one of several typical nonlinear transformations of the sum:

$$neuron_{ij}(\boldsymbol{x}) = nonlinearity(\boldsymbol{w}_{ij}^T \boldsymbol{x} + w_{ij,0})$$

the weights w are parameters of the neuron

$$nonlinearity(y) = \frac{1}{1+\exp(-y)}$$

logistic nonlinearity

$$nonlinearity(y) = \tanh(y)$$

hyperbolic tangent nonlinearity

$$nonlinearity(y) = max(0, y)$$

"rectified linear unit" nonlinearity

Recurrent networks

- In feedforward neural networks the input of each neuron are the set of outputs of the previous layers: the input for f5 is f4, the input for f4 is f3, the input for f3 is f2, the input for f2 is f1, and the input for f1 is the input text.
- However, this means that if e.g. the input is a window of 10 words, whatever happened earlier than those 10 words does not affect the network output predictions
- Recurrent networks use neurons that have a "memory": at time t, their own previous outputs at time t-1 are used as another input

$$neuron_{ij}(\mathbf{x}_t) = nonlinearity(\mathbf{w}_{ij}^T [\mathbf{x}, neuron_{ij}(\mathbf{x}_{t-1})] + w_{ij,0})$$

- More generally, the neuron can depend on previous outputs of the entire layer

$$neuron_{ij}(\mathbf{x}_t) = nonlinearity(\mathbf{w}_{ij}^T [\mathbf{x}, neuron_{i1}(\mathbf{x}_{t-1}), \ldots, neuron_{iK}(\mathbf{x}_{t-1})] + w_{ij,0})$$

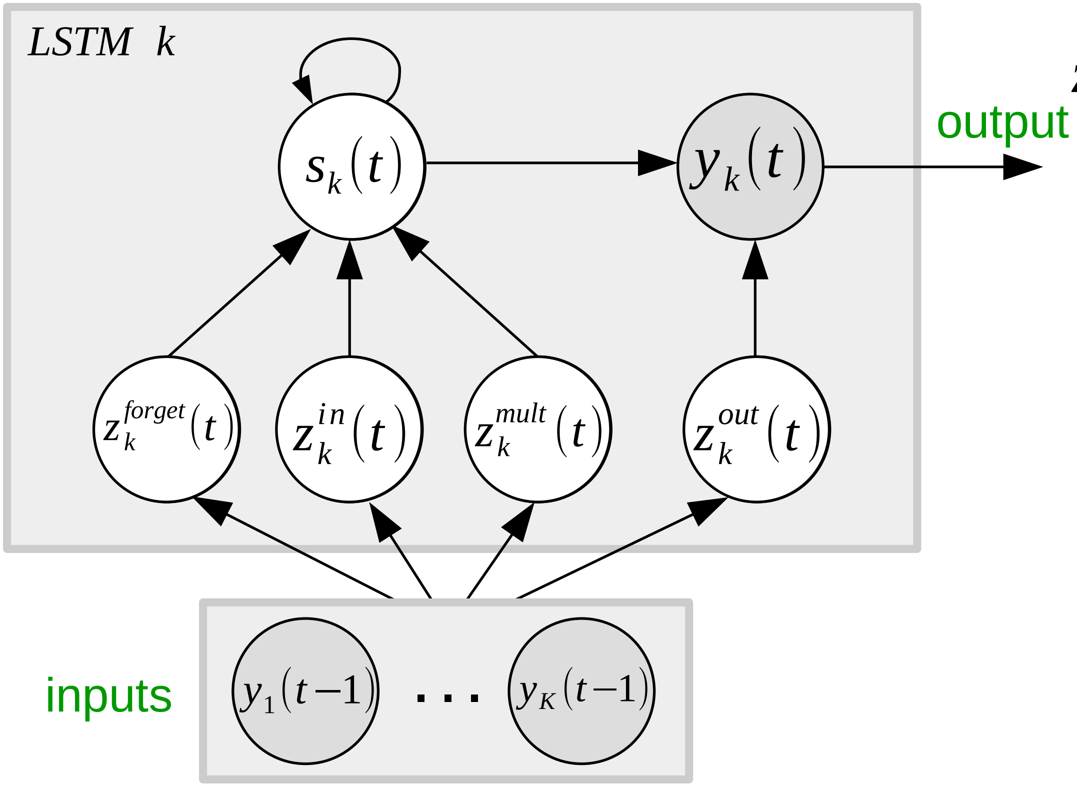
- Various other architectures allow connections to previous outputs of other layers too.

LSTM (long short-term memory)

- References:
  - S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation* 9 (8), 1735–1780, 1997.
  - F. A. Gers. Learning to Forget: Continual Prediction with LSTM. Proc. ICANN 1999, pages 850–855, 1999.
  - F. A. Gers, N. N. Schraudolph, and J. Schmidhuber. Learning Precise Timing with LSTM Recurrent Networks", Journal of Machine Learning Research 3, 115–143, 2002.
  - Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM Neural Networks for Language Modeling. In Proc. Interspeech 2012, 2012.
- Idea: replace the usual recurrent neuron by a more complicated unit that involves several nonlinearities g and h

LSTM (long short-term memory)

- Mathematics:



$$z_k^{forget}(t) = g^{forget}\left(\sum_u w_k^{forget} y_u(t-1)\right)$$

$$z_k^{in}(t) = g^{in}\left(\sum_u w_k^{in} y_u(t-1)\right)$$

$$z_k^{mult}(t) = g^{mult}\left(\sum_u w_k^{mult} y_u(t-1)\right)$$

$$z_k^{out}(t) = g^{out}\left(\sum_u w_k^{out} y_u(t-1)\right)$$

$$s_k(t) = s_k(t-1)\, z_k^{forget}(t) + z_k^{\text{in}}(t)\, z_k^{mult}(t)$$

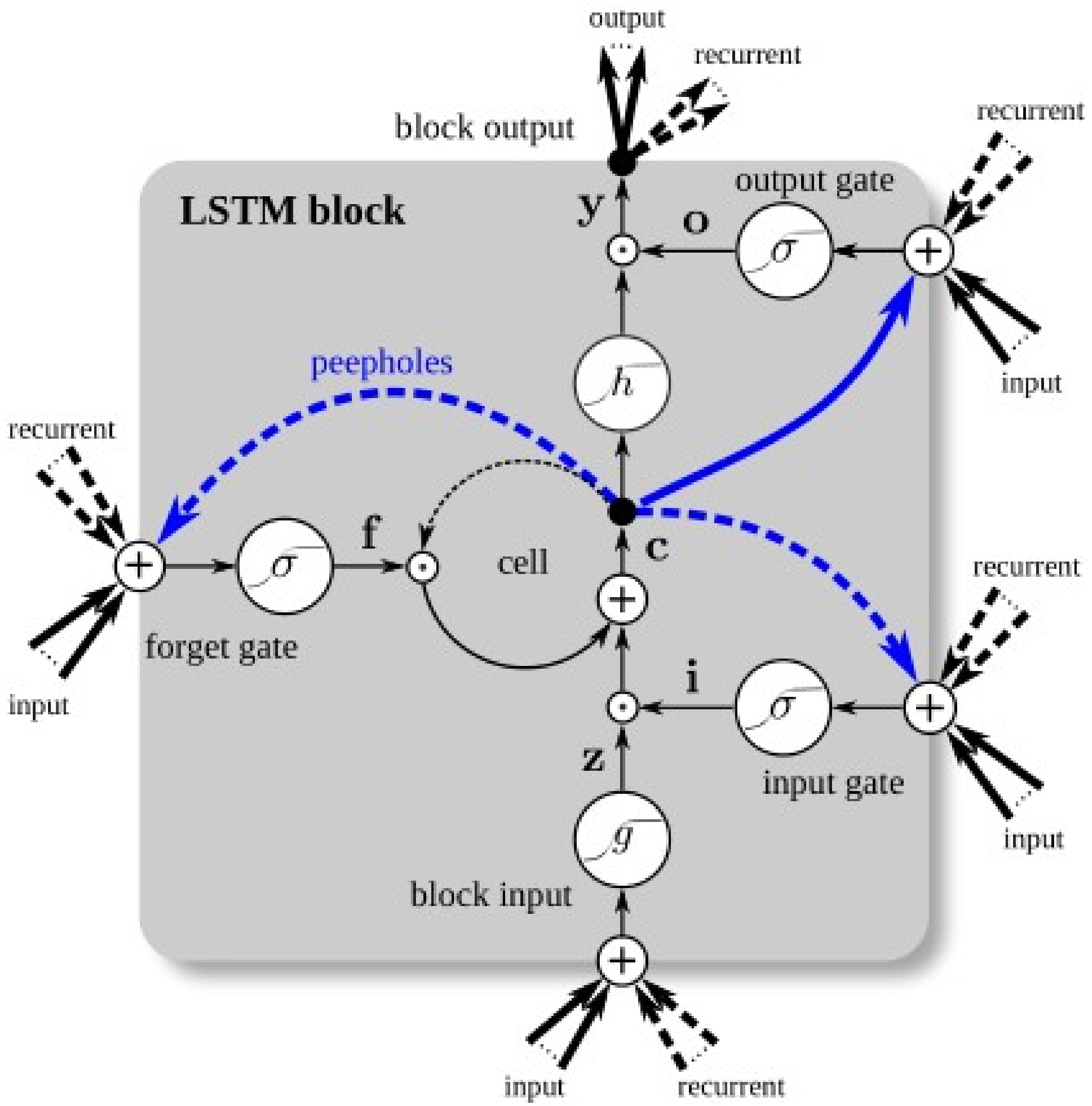
$$y_k(t) = s_k(t)\, h\left(z_k^{out}(t)\right)$$

LSTM (long short-term memory)

- LSTM in more detail.
- Neural networks suffer from a **vanishing gradient problem**. In a function like **f** = **f**5(**f**4(**f**3(**f**2(**f**1(inputs))))), derivatives of a cost function with respect to parameters of f1 must be computed through multiplication of derivatives of f5 with respect to f4, f4 w.r.t. f3, f3 w.r.t. f2, f2 w.r.t. f1, and finally f1 w.r.t. its parameters.
- The multiplied derivatives can easily either decay to zero or grow exponentially.
- In a **recurrent network**, values of f1-f5 from a previous time step affect the output: **f**=**f**5(**f**4(**f**3(**f**2(**f**1(inputs,prev.f1),prev.f2),prev.f3),prev.f4),prev.f5), this means the vanishing gradient problem could cause the gradient to decay to zero or grow exponentially **over time**.
- The LSTM architecture was designed to avoid the vanishing gradient problem over time.
- The idea is to fix the "scaling factor" of individual network units (neurons) to 1, but provide additional complexity by **gating functions** within the unit.

LSTM (long short-term memory)

- Mathematics:

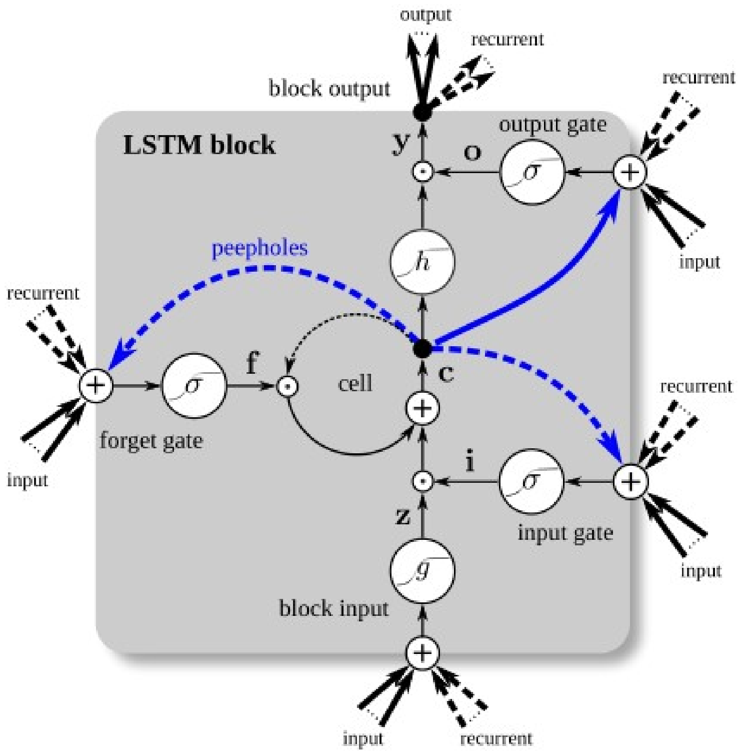


From LSTM: A Search Space Odyssey, Klaus Greff, Rupesh K. Srivastava, Jan Koutn´ ık, Bas R. Steunebrink, Jürgen Schmidhuber, IEEE TNNLS, 2016.

LSTM (long short-term memory)

- Mathematics:

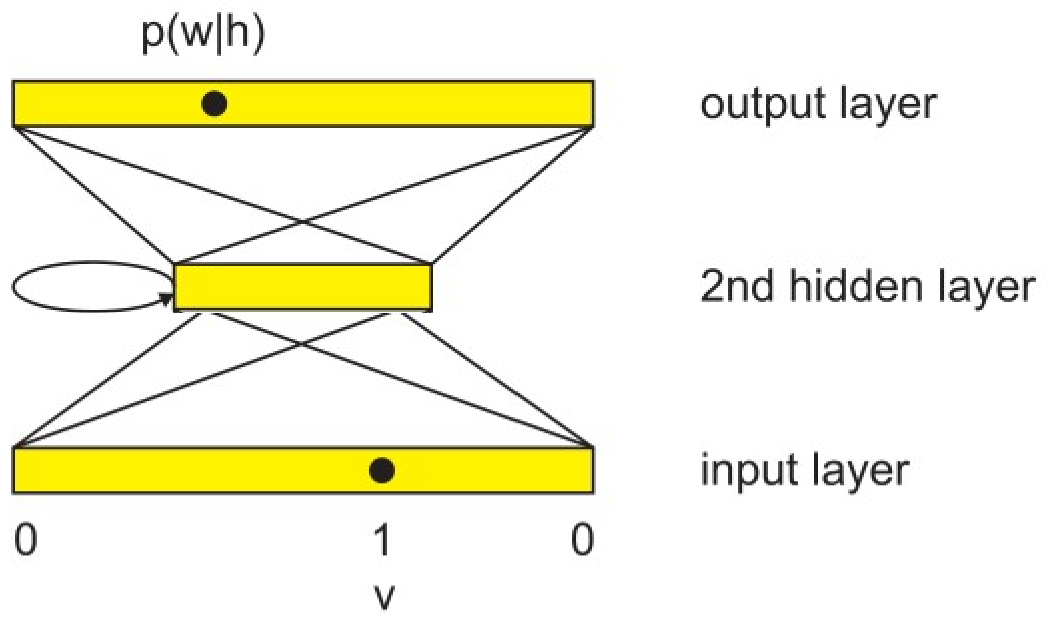
From LSTM: A Search Space Odyssey, Klaus Greff, Rupesh K. Srivastava, Jan Koutn´ ık, Bas R. Steunebrink, Jürgen Schmidhuber, IEEE TNNLS, 2016.



$$
\begin{aligned}
\bar{\mathbf{z}}^t &= \mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z \\
\mathbf{z}^t &= g(\bar{\mathbf{z}}^t) && \textit{block input} \\
\bar{\mathbf{i}}^t &= \mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i \\
\mathbf{i}^t &= \sigma(\bar{\mathbf{i}}^t) && \textit{input gate} \\
\bar{\mathbf{f}}^t &= \mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f \\
\mathbf{f}^t &= \sigma(\bar{\mathbf{f}}^t) && \textit{forget gate} \\
\mathbf{c}^t &= \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t && \textit{cell} \\
\bar{\mathbf{o}}^t &= \mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o \\
\mathbf{o}^t &= \sigma(\bar{\mathbf{o}}^t) && \textit{output gate} \\
\mathbf{y}^t &= h(\mathbf{c}^t) \odot \mathbf{o}^t && \textit{block output}
\end{aligned}
$$

LSTM (long short-term memory)

- Example architecture of full network:



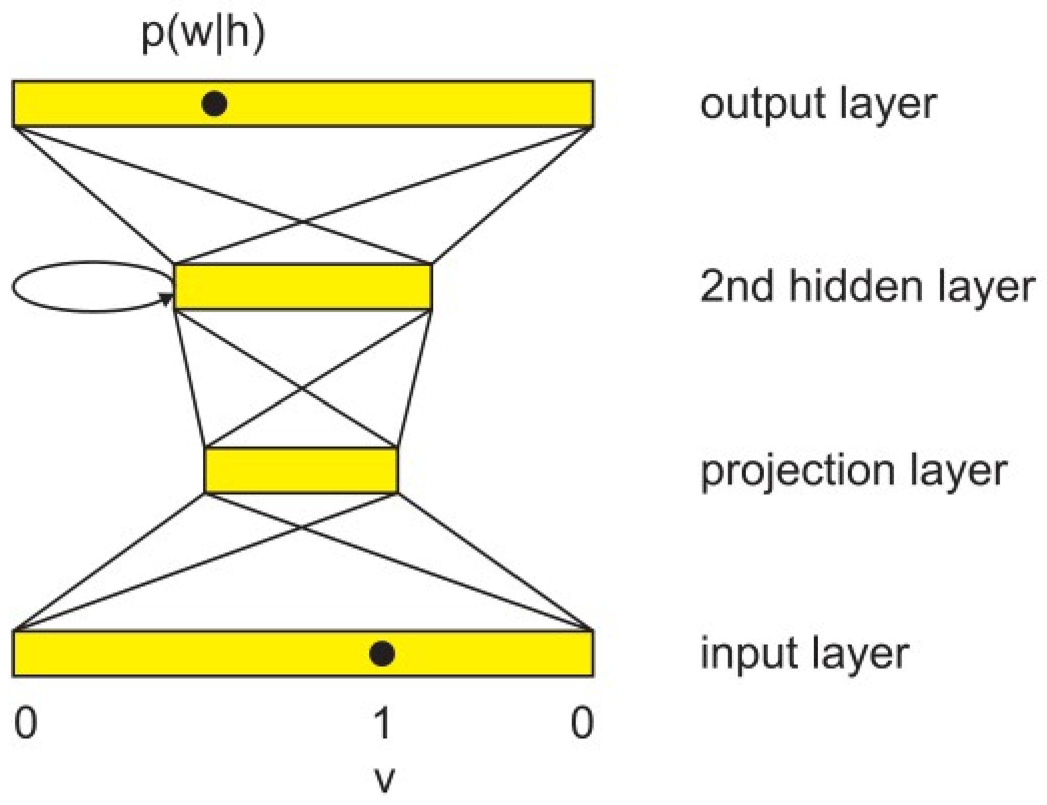
- The **output layer** computes probability of each possible word in the vocabulary, using softmax transformation of affinities (**f2**) to the previous layer output.
- The **hidden layer** (**f1**) is composed of LSTM units. Their output depends on affinities to the inputs, and on their previous outputs, and their previous internal values.
- The **input layer** is composed of a window of words, each encoded as a binary one-out-of-V vector (**inputs**)

Resulting prediction function:
p(w|inputs) =
softmax (**f2** (**f1**(**inputs, prev.f1**)))

Picture based on Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM Neural Networks for Language Modeling. In Proc. Interspeech 2012, 2012.

LSTM (long short-term memory)

- Example architecture of full network:



Picture from Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM Neural Networks for Language Modeling. In Proc. Interspeech 2012, 2012.

- The **output layer** computes probability of each possible word in the vocabulary, using softmax transformation of affinities to the previous layer output (**f3**).
- The **2nd hidden layer** (**f2**) is composed of LSTM units. Their output depends on affinities to the previous layer output, and on their previous outputs, and their previous internal values.
- The **projection layer** (**f1**) is composed of normal neurons. Their output depends on affinities to the input.
- The **input layer** is composed of a window of words, each encoded as a binary one-out-of-V vector (**inputs**)

Resulting prediction function:
p(w|inputs) =
softmax (**f3**(**f2** (**f1**(**inputs**), **prev.f2**)))

LSTM (long short-term memory)

- Observations from experiments of Sundermeyer et al.:
- English Treebank-3 Corpus (930k training words, 10k vocabulary size) and a French corpus from the Quaero 2 project (27M words, 170k vocabulary size)
- Performance measure: log-likelihood of held-out test set (ability to predict content of data not seen in training)
  - Actually they use perplexity = exp(-1 * log-likelihood), smaller is better.
- Perplexity of LSTM models was lower by about 8% compared to a standard recurrent neural network
- A small improvement is possible using two concatenated sentences instead of one
- Using a projection layer in LSTM:
  - caused negligible additional training time,
  - For 1-sentence inputs results did not improve compared to the only 1 LSTM layer, regardless whether the projection layer used linear neurons (no nonlinearity) or sigmoidal nonlinearities.
  - For 2-sentence inputs, a linear projection layer improved results
  - Their analysis: projection "smeared" input features and complicated LSTM learning

LSTM (long short-term memory)

- In Python, LSTM networks can be used with the **tensorflow** and **keras** libraries.

```
pip install tensorflow
pip install keras
import keras
import keras.models
import keras.layers
import keras.callbacks
import keras.utils
import keras.utils.np_utils
```

LSTM (long short-term memory)

- In Python, LSTM networks can be used with the **tensorflow** and **keras** libraries.

```
# Create the network model
networkmodel = keras.models.Sequential()
# Add the first layer after the input layer (here a LSTM
layer)
lstm_layer_size=50
text_windowsize=10
features_per_input_sample=1
networkmodel.add(keras.layers.LSTM(\
    lstm_layer_size, activation='relu', \
    input_shape=(text_windowsize,\
    features_per_input_sample)))
# Add the output layer
output_layer_size=len(unifiedvocabulary)
networkmodel.add(keras.layers.Dense(output_layer_size,\
    activation='softmax'))
networkmodel.compile(loss='categorical_crossentropy',\
    optimizer='adam',metrics=['accuracy'])
```

LSTM (long short-term memory)

- In Python, LSTM networks can be used with the **tensorflow** and **keras** libraries.

```
# Create a training data set: each sample is a
# window of input words and a 1-out-of-V encoded
# vector for the output word
n_trainingsamples=len(mycrawled_lowercasetexts[0])-
text_windowsize
x_data=numpy.zeros((n_trainingsamples,text_windowsize,1))
y_data=numpy.zeros((n_trainingsamples))
temp_wordindices=myindices_in_unifiedvocabulary[0]
for windowposition in range(n_trainingsamples):
    for k in range(text_windowsize):
        temp_wordindex=temp_wordindices[windowposition+k]
        x_data[windowposition,k,0]=temp_wordindex
    temp_wordindex=\
        temp_wordindices[windowposition+text_windowsize]
    y_data[windowposition]=temp_wordindex
y_onehotencoded=keras.utils.to_categorical(y_data,\
    num_classes=len(unifiedvocabulary))
```

LSTM (long short-term memory)

- In Python, LSTM networks can be used with the **tensorflow** and **keras** libraries.

```
# Fit the model
networkmodel.fit(x=x_data,\
    y=y_onehotencoded,\
    batch_size=100,epochs=10)
```

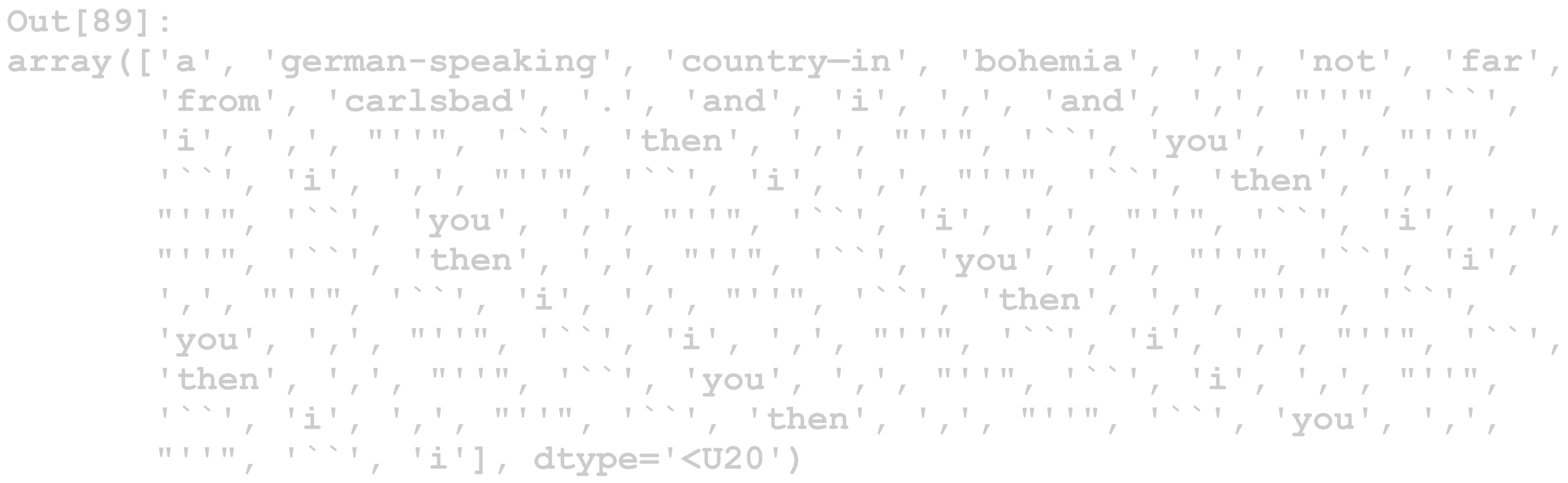
LSTM (long short-term memory)

- In Python, LSTM networks can be used with the **tensorflow** and **keras** libraries.

```
# Predict new text
n_newtext=100
start_textwindow=2000
generated_wordindices=numpy.squeeze(\
    numpy.array(x_data[start_textwindow,:,:]))
for k in range(n_newtext):
    temp_textwindow=generated_wordindices[k:(k+text_windowsize)]
    temp_probs=networkmodel.predict(numpy.reshape(\
        temp_textwindow,(1,text_windowsize,1)))
    best_word=numpy.argmax(temp_probs)
    generated_wordindices=numpy.append(\
        generated_wordindices,best_word)
generated_wordindices=generated_wordindices.astype(int)
unifiedvocabulary[generated_wordindices]
```

LSTM (long short-term memory)

- In Python, LSTM networks can be used with the **tensorflow** and **keras** libraries.



```
Out[89]:
array(['a', 'german-speaking', 'country—in', 'bohemia', ',', 'not', 'far',
       'from', 'carlsbad', '.', 'and', 'i', ',', 'and', ',', "''", '``',
       'i', ',', "''", '``', 'then', ',', "''", '``', 'you', ',', "''",
       '``', 'i', ',', "''", '``', 'i', ',', "''", '``', 'then', ',',
       "''", '``', 'you', ',', "''", '``', 'i', ',', "''", '``', 'i', ',',
       "''", '``', 'then', ',', "''", '``', 'you', ',', "''", '``', 'i',
       ',', "''", '``', 'i', ',', "''", '``', 'then', ',', "''", '``',
       'you', ',', "''", '``', 'i', ',', "''", '``', 'i', ',', "''", '``',
       'then', ',', "''", '``', 'you', ',', "''", '``', 'i', ',', "''",
       '``', 'i', ',', "''", '``', 'then', ',', "''", '``', 'you', ',',
       "''", '``', 'i'], dtype='<U20')
```

- Not yet great, may need more training than just 10 epochs.
- For a quicker result, let's try with characters!

LSTM (long short-term memory)

```
# Create a list of each character in order of appearance
temptext=' '.join(mycrawled_lowercasetexts[0])
temptext=list(temptext)
# Create the "vocabulary" of the individual characters
tempvocabulary=[]
myindices_in_tempvocabulary=[]
uniqueresults=numpy.unique(temptext,return_inverse=True)
charactervocabulary=uniqueresults[0]
characterindices=uniqueresults[1]
```

LSTM (long short-term memory)

```
# Create the network model
networkmodel = keras.models.Sequential()
# Add the first layer after the input layer
#  (here a LSTM layer)
lstm_layer_size=50
character_windowsize=50
features_per_input_sample=1
networkmodel.add(keras.layers.LSTM(\
    lstm_layer_size, activation='relu', \
    input_shape=(character_windowsize,\
    features_per_input_sample)))
# Add the output layer
output_layer_size=len(charactervocabulary)
networkmodel.add(keras.layers.Dense(output_layer_size,\
    activation='softmax'))
networkmodel.compile(loss='categorical_crossentropy',\
    optimizer='adam',metrics=['accuracy'])
```

LSTM (long short-term memory)

```
# Create a training data set: each sample is a window of
# input characters and a 1-out-of-V encoded vector for the
# output character
n_trainingsamples=len(characterindices)-character_windowsize
x_data=numpy.zeros((n_trainingsamples,character_windowsize,1))
y_data=numpy.zeros((n_trainingsamples))
for windowposition in range(n_trainingsamples):
    for k in range(character_windowsize):
        temp_characterindex=characterindices[windowposition+k]
        x_data[windowposition,k,0]=temp_characterindex
    temp_characterindex=\
        characterindices[windowposition+character_windowsize]
    y_data[windowposition]=temp_characterindex
y_onehotencoded=keras.utils.to_categorical(y_data,\
    num_classes=len(charactervocabulary))

# Fit the model
networkmodel.fit(x=x_data,y=y_onehotencoded,\
    batch_size=100,epochs=10)

# Accuracy after about 15min training: 0.4149
```

LSTM (long short-term memory)

```
# Predict new text
n_newtext=100
start_characterwindow=2000
generated_characterindices=numpy.squeeze(\
    numpy.array(x_data[start_characterwindow,:,:]))
for k in range(n_newtext):
    temp_characterwindow=\
        generated_characterindices[k:(k+character_windowsize)]
    temp_probs=networkmodel.predict(numpy.reshape(\
        temp_characterwindow,(1,character_windowsize,1)))
    best_character=numpy.argmax(temp_probs)
    generated_characterindices=numpy.append(\
         generated_characterindices,best_character)
generated_characterindices=\
    generated_characterindices.astype(int)


''.join(charactervocabulary[generated_characterindices])

Out[124]: "of society with his whole bohemian soul ,
remained the soon . '' `` i have he had been the soace ,
and i have been the soace , and i have been the soa"
```

LSTM (long short-term memory)

```
# Predict new text, with sampling
n_newtext=100
start_characterwindow=2000
generated_characterindices=numpy.squeeze(\
    numpy.array(x_data[start_characterwindow,:,:]))
for k in range(n_newtext):
    temp_characterwindow=\
        generated_characterindices[k:(k+character_windowsize)]
    temp_probs=networkmodel.predict(numpy.reshape(\
        temp_characterwindow,(1,character_windowsize,1)))
    #best_character=numpy.argmax(temp_probs)
    temp_probs=numpy.squeeze(numpy.array(temp_probs))
    best_character=numpy.random.choice(}
        range(len(charactervocabulary)), p=temp_probs)
    generated_characterindices=numpy.append(\
        generated_characterindices,best_character)
generated_characterindices=generated_characterindices.astype(int)
''.join(charactervocabulary[generated_characterindices])
```

```
Out[162]: 'of society with his whole bohemian soul , remained . tie sabill-. au
fre the beslnn tuadeos in . tian vhen yould fold fyaytedfe . au a ytrds fyrbnldr
'
Out[163]: 'of society with his whole bohemian soul , remained bnalriy ph a ber-
dakd foro lfc of tie gveipee the biyiny wouhdf ohgcser eyrrinyselly tomnge ciaib
o'
```

LSTM (long short-term memory)

- Instead of using the 1-out-of-V word indices as input, one could use an embedding,
  - either by using a projection (Embedding) layer in the network,
  - or by providing the input words with predefined features: e.g. computed by word2vec on the same data, or a pretrained embedding
- In python, keras allows to add an embedding layer as the first layer:

```
output_dimensionality=20
networkmodel.add(keras.layers.Embedding(\
     len(unifiedvocabulary),\
     output_dimensionality, input_length=textwindow_size))
```

- Several other architectures (combinations of layers) can be tried

Other neural models

- Transformers: alternatives to recurrent neural networks. They involve an encoder-decoder architecture with attention mechanisms. The building blocks are scaled dot-product attention units
- BERT models: Bidirectional Encoder Representations from Transformers, a pre-trained transformer model
- GPT models: Generative Pre-trained Transformer