Language Technology
http://cs.lth.se/edan20/
Sequence Tagging Using Machine-Learning Techniques

Pierre Nugues

Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

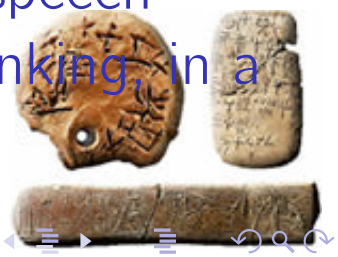
September 21 and 28, 2023

Motivation

The analysis of sentences often involves the analysis of words or groups of words (chunks).
Three related tasks:

1. Identify the type of word, for instance noun or verb using the classical grammar:
   The **waiter** *brought the* **meal**
2. Identify groups or segments, noun groups for instance:
   The waiter *brought* **the meal**
3. Identify a name (a proper noun) for instance are these three words,
   Kjell Olof Andersson, *the waiter*

This lecture will show you how to solve the first one, part-of-speech tagging, and you will write a program for the second one, chunking, in a next laboratory assignment.



Model

We can model the problem as the conversion of an input sequence to an output

Output:	**y**	DET	NOUN	VERB	DET	NOUN
		↑	↑	↑	↑	↑
Input:	**x**	The	waiter	brought	the	meal

Part-of-Speech Annotation (CoNLL 2000)

Annotation of: *He reckons the current account deficit will narrow to only # 1.8 billion in September.* We set aside the last column for now.

He	PRP	B-NP
reckons	VBZ	B-VP
the	DT	B-NP
current	JJ	I-NP
account	NN	I-NP
deficit	NN	I-NP
will	MD	B-VP
narrow	VB	I-VP
to	TO	B-PP
only	RB	B-NP
#	#	I-NP
1.8	CD	I-NP
billion	CD	I-NP
in	IN	B-PP
September	NNP	B-NP
.	.	O

Designing a Part-of-Speech Tagger

We will now create part-of-speech taggers
No unique solution
We will examine three architectures:

1. A feed-forward pipeline with a one-hot encoding of the words;
2. A feed-forward pipeline with word embeddings: We will replace the one-hot vectors with GloVe embeddings;
3. A recurrent neural network, either a simple RNN or a LSTM, with word embeddings.

Features for Part-of-Speech Tagging

The word *visit* is ambiguous in English:

I paid a **visit** *to a friend –> noun*

I went to **visit** *a friend –> verb*

The context of the word enables us to tell, here an article or the infinitive marker

To train and apply the model, the tagger extracts a set of features from the surrounding words, for example, a sliding window spanning five words and centered on the current word.

We then associate the feature vector $(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2})$ with the part-of-speech tag t_i at index i.

Training Set

Part-of-speech taggers use a training set where every word is hand-annotated (Penn Treebank and CoNLL 2008).

Index	Word	Hand annotation	Index	Word	Hand annotation
1	Battle	JJ	19	of	IN
2	-	HYPH	20	their	PRP$
3	tested	JJ	21	countrymen	NNS
4	Japanese	JJ	22	to	TO
5	industrial	JJ	23	visit	VB
6	managers	NNS	24	Mexico	NNP
7	here	RB	25	,	,
8	always	RB	26	a	DT
9	buck	VBP	27	boatload	NN
10	up	RP	28	of	IN
11	nervous	JJ	29	samurai	FW
12	newcomers	NNS	30	warriors	NNS
13	with	IN	31	blown	VBN
14	the	DT	32	ashore	RB
15	tale	NN	33	375	CD
16	of	IN	34	years	NNS
17	the	DT	35	ago	RB
18	first	JJ	36	.	.

Architecture 1: Part-of-Speech Tagging with Linear Classifiers

Linear classifiers are efficient devices to carry out part-of-speech tagging:

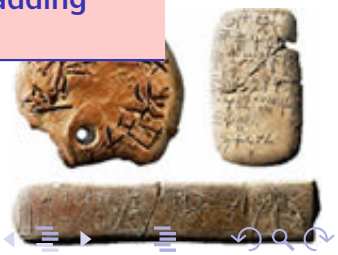
1. The lexical values are the input data to the tagger.
2. The parts of speech are assigned from left to right by the tagger.

ID	FORM	PPOS	
	BOS	BOS	Padding
	BOS	BOS	
1	Battle	NN	
2	-	HYPH	
3	tested	NN	
...	...	...	
17	the	DT	
18	first	JJ	
19	of	IN	
20	their	PRP$	
21	countrymen	NNS	Input features
22	to	TO	
23	**visit**	VB	Predicted tag
24	Mexico		↓
25	,		
26	a		
27	boatload		
...	...	...	
34	years		
35	ago		
36	.		
	EOS		Padding
	EOS		

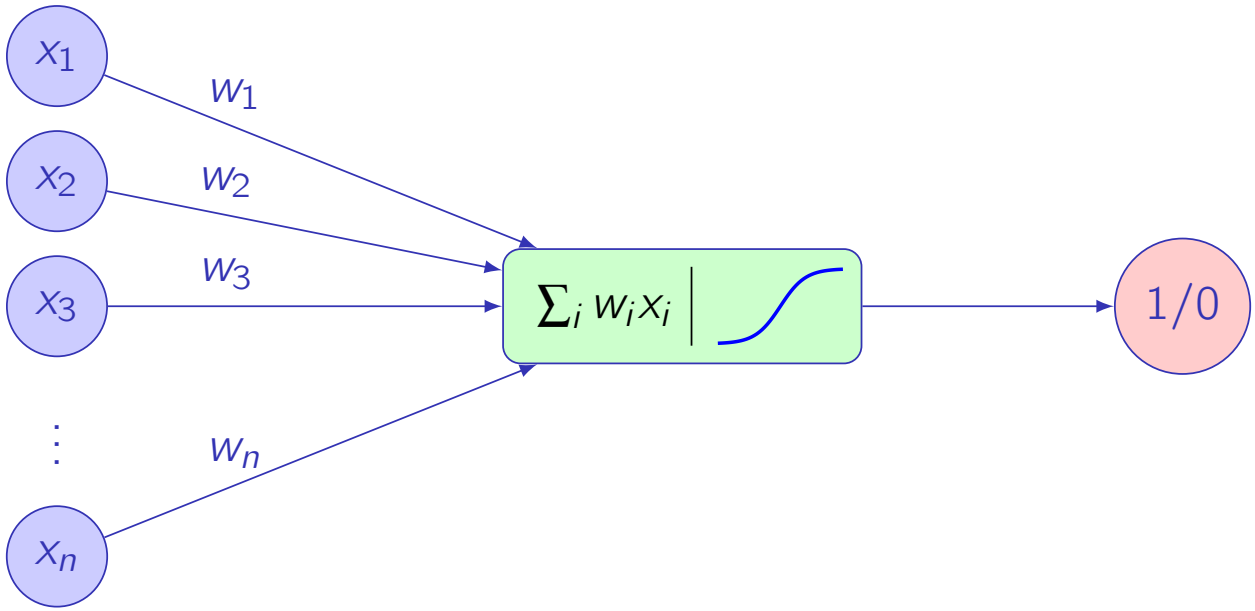
Feed-Forward Structure

As input, the classifier uses: $(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2})$ to predict the part-of-speech tag t_i at index i.
Here:
(countrymen, to, visit, Mexico, ",")
to predict VB

ID	FORM	PPOS	
	BOS	BOS	**Padding**
	BOS	BOS	
1	Battle	NN	
2	-	HYPH	
3	tested	NN	
...	...	...	
17	the	DT	
18	first	JJ	
19	of	IN	
20	their	PRP$	
21	countrymen	NNS	**Input features**
22	to	TO	
23	**visit**	VB	**Predicted tag**
24	Mexico		↓
25	,		
26	a		
27	boatload		
...	...	...	
34	years		
35	ago		
36	.		
	EOS		**Padding**
	EOS		

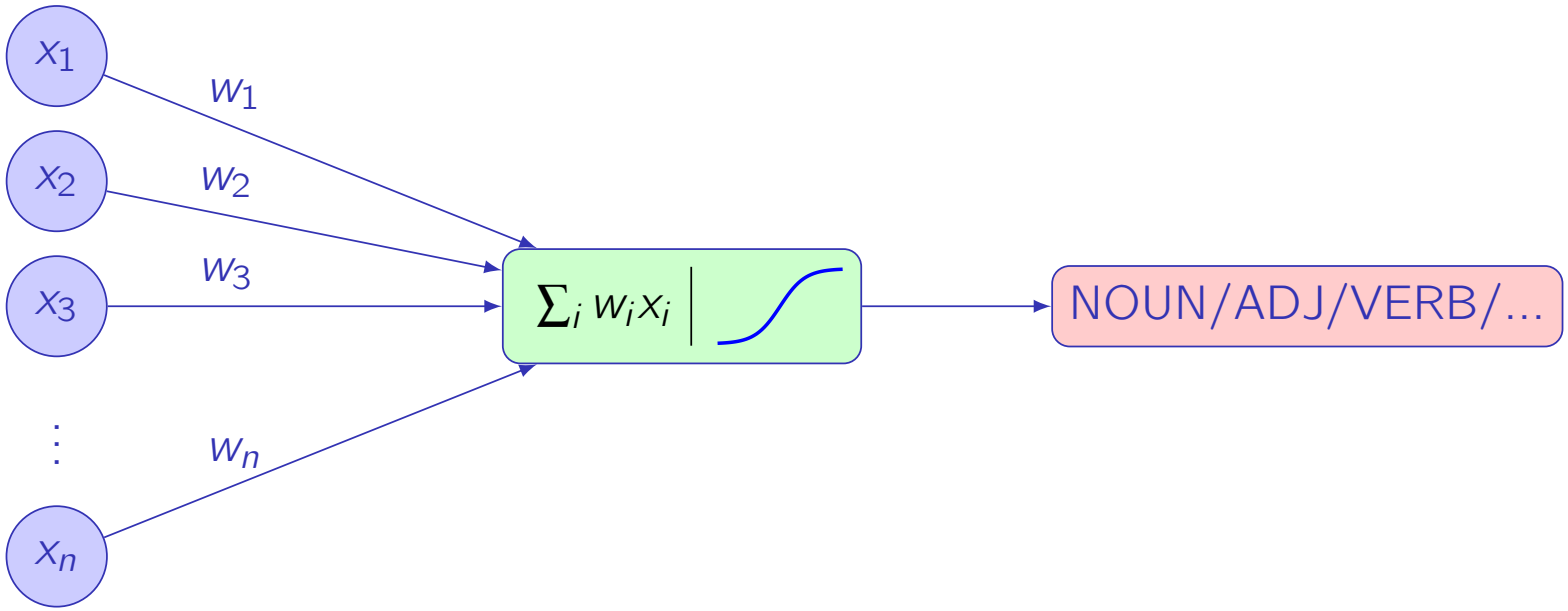


Feed Forward (Binary)



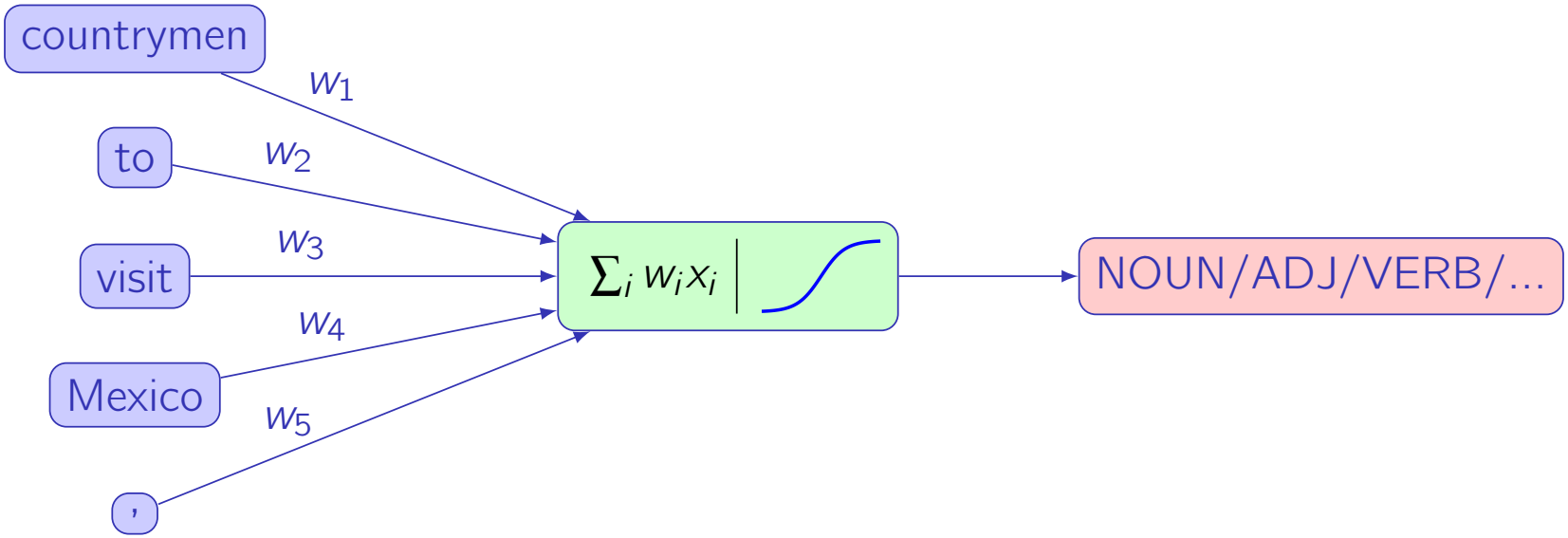
Logistic function

Feed Forward (Multinomial)



Softmax

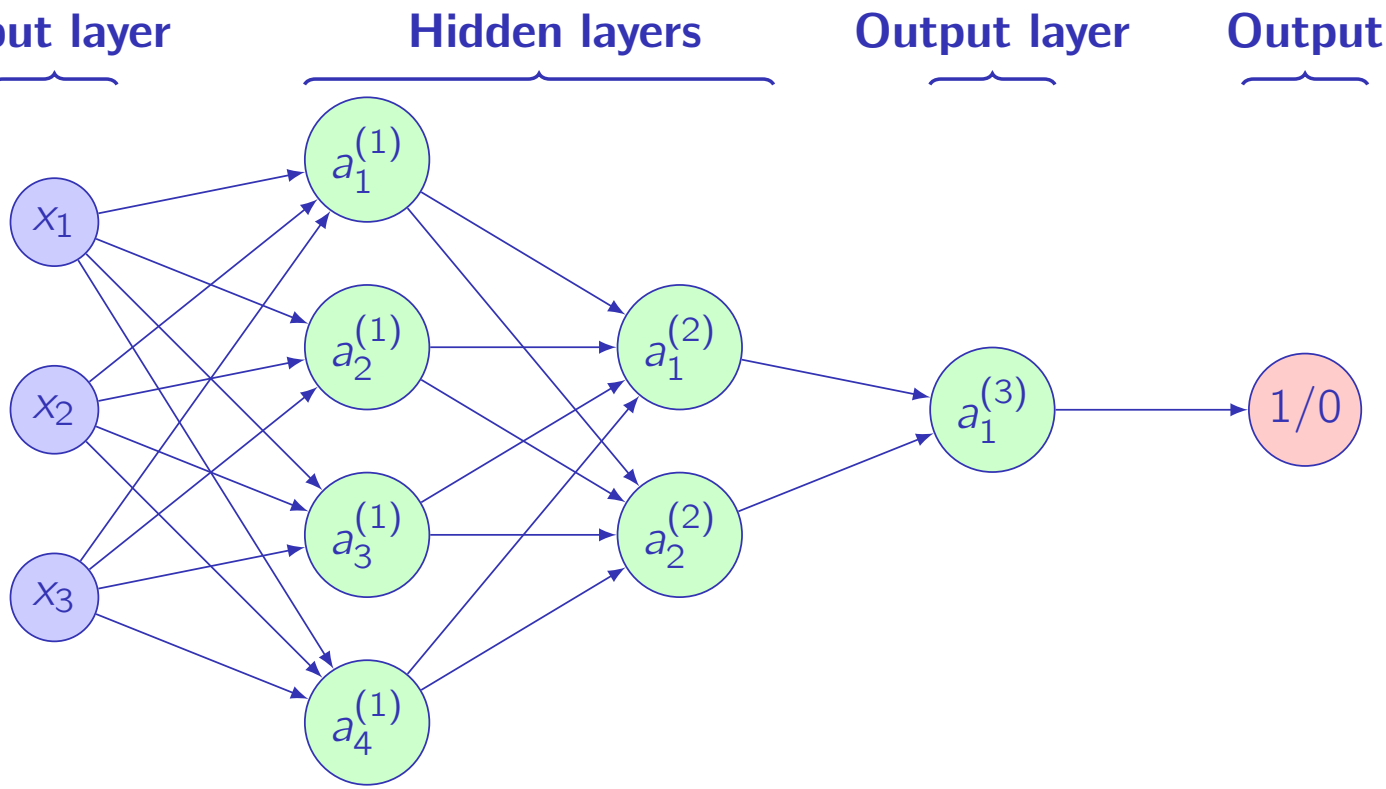
Feed Forward (Multinomial) (II)



Input: one-hot encoding

Output: Softmax to predict the parts of speech

Feed Forward (Multilayer)



For the first layer, we have:

$$\text{activation}(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}).$$

Feature Vectors

ID	Feature vectors: X					PPOS: $\hat{y}$
	w_{i-2}	w_{i-1}	w_i	w_{i+1}	w_{i+2}	
1	BOS	BOS	Battle	-	tested	NN
2	BOS	Battle	-	tested	Japanese	HYPH
3	Battle	-	tested	Japanese	industrial	JJ
...	...	...	...	...	...	...
19	the	first	of	their	countrymen	IN
20	first	of	their	countrymen	to	PRP$
21	of	their	countrymen	to	visit	NNS
22	their	countrymen	to	visit	Mexico	TO
23	countrymen	to	visit	Mexico	,	VB
24	to	visit	Mexico	,	a	NNP
25	visit	Mexico	,	a	boatload	,
...	...	...	...	...	...	...
34	ashore	375	years	ago	.	NNS
35	375	years	ago	.	EOS	RB
36	years	ago	.	EOS	EOS	.

Word Encoding: One-hot encoding

The feature space is defined by all the word values and a word has one dimension: a unit vector

Encoding with unit vectors yields a sparse representation

We use `DictVectorizer()` to encode them:

```
from sklearn.feature_extraction import DictVectorizer
v = DictVectorizer(sparse=False)
X_cat = [{'w_1': 'the', 'w_2': 'first', 'w_3': 'of'},
         {'w_1': 'first', 'w_2': 'of', 'w_3': 'the'},
         {'w_1': 'of', 'w_2': 'the', 'w_3': 'countrymen'},
         {'w_1': 'the', 'w_2': 'countrymen', 'w_3': 'to'}]
X = v.fit_transform(X_cat)
X
array([[0., 0., 1., 0., 1., 0., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 0., 1., 1., 0., 0., 0.],
       [0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 1.]])
```

Word Encoding

```
v.transform([{'w_1': 'the', 'w_2': 'of', 'w_3': 'Mexico'}])
array([[0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0.]])

v.get_feature_names()
['w_1=first',
 'w_1=of',
 'w_1=the',
 'w_2=countrymen',
 'w_2=first',
 'w_2=of',
 'w_2=the',
 'w_3=countrymen',
 'w_3=of',
 'w_3=the',
 'w_3=to']
```

A Feed-Forward Neural Network with PyTorch

We first use a feed-forward architecture corresponding to a logistic regression.
Here we use a logit output (no activation for the last layer.)

```
if SIMPLE_MODEL:
    model = nn.Sequential(nn.Linear(X_train.size()[1],
                                    NB_CLASSES))
else:
    model = nn.Sequential(
        nn.Linear(X_train.size()[1],
                  NB_CLASSES * 2),
        nn.ReLU(),
        nn.Dropout(0.2),
        nn.Linear(NB_CLASSES * 2, NB_CLASSES))
```

Preprocessing

Preprocessing is more complex though: Four steps:

1. Read the corpus

```
train_sentences, dev_sentences, test_sentences, \
  column_names = load_ud_en_ewt()
```

2. Store the rows of the CoNLL corpus in dictionaries

```
conll_dict = CoNLLDictorizer(column_names, col_sep='\t')
train_dict = conll_dict.transform(train_sentences)
test_dict = conll_dict.transform(test_sentences)
```

3. Extract the features and store them in dictionaries

```
context_dictorizer = ContextDictorizer()
context_dictorizer.fit(train_dict)
X_dict, y_cat = context_dictorizer.transform(train_dict)
```

4. Vectorize the symbols

```
# We transform the X symbols into numbers
dict_vectorizer = DictVectorizer()
X_num = dict_vectorizer.fit_transform(X_dict)
```

Code Example

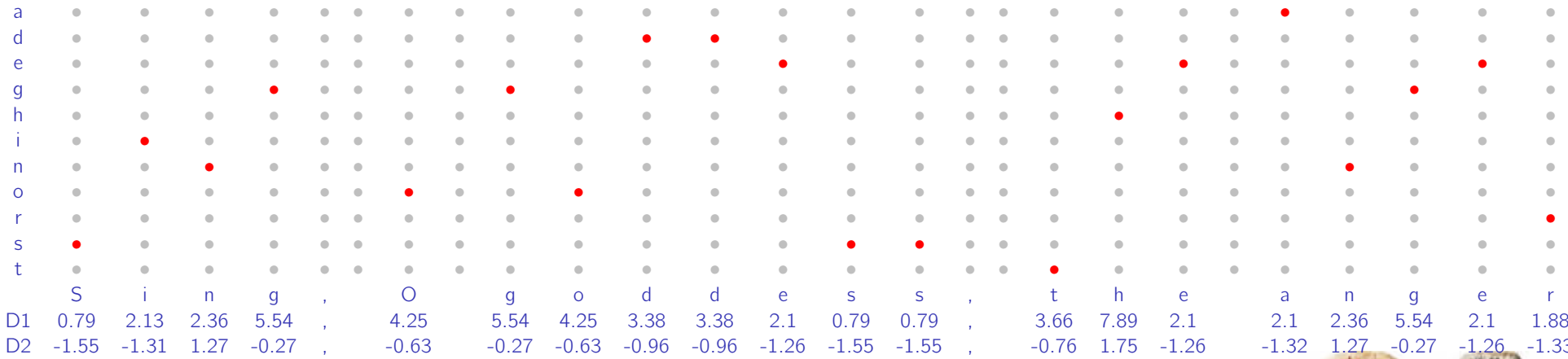
Jupyter Notebook: `ch10-pos-tagger_ff.ipynb`

Architecture 2: Dense Word Encoding

One-hot encoding is results in sparse matrices

An alternative is to use dense vectors, for instance from a principal component analysis. Compare the character encodings of:

Sing, O Goddess, the anger...



Using Embeddings

We replace the one-hot vectors with embeddings, the rest being the same
Word embeddings are dense vectors obtained by a principal component analysis or another method.
They can be trained by the neural network or pretained
In this implementation:

1. We use pretrained embeddings from the GloVe project;
2. Our version of GloVe is lowercased, so we set all the characters in lowercase;
3. We add the embeddings as an `Embedding` layer at the start of the network;
4. We initialize the embedding layer with GloVe and make it trainable or not.

It would be possible to use a randomly initialized matrix as embeddings instead

The Model with Embeddings

Outline to create the embedding matrix:

```
embedding_matrix = np.random.random(
                                (len(vocabulary_words) + 1,
                                EMBEDDING_DIM))


for word in vocabulary_words:
    if word in embeddings_dict:
        embedding_matrix[word_idx[word]] = embeddings_dict[word]
```

The model:

```
model = nn.Sequential(
    nn.Embedding.from_pretrained(embedding_matrix, freeze=False),
    nn.Flatten(),
    nn.Linear(5 * embedding_matrix.size()[1], NB_CLASSES)
)
```

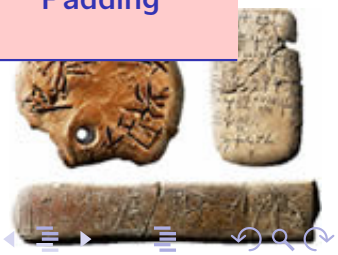
Code Example

Jupyter Notebook: `ch10-pos-tagger_ff_embs.ipynb`

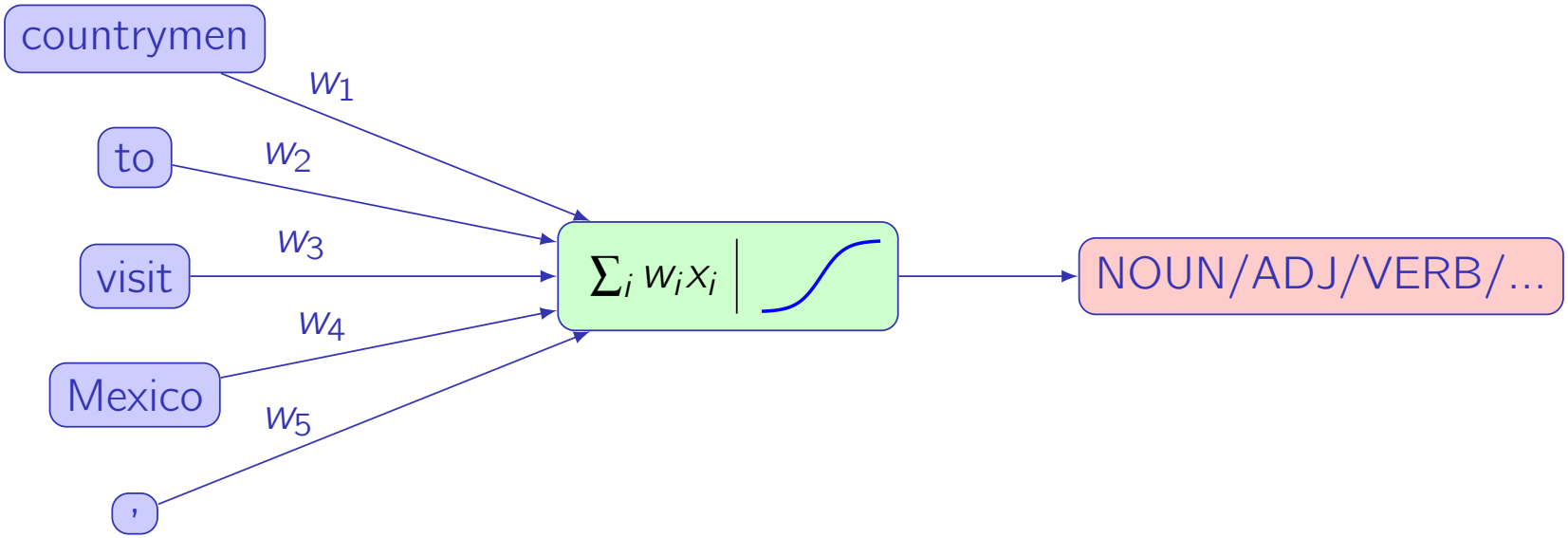
From Feed-Forward to Recurrent

- As input, the classifier uses: $(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2})$ to predict the part-of-speech tag t_i at index i.
- Here: (countrymen, to, visit, Mexico, ",") to predict VB

ID	FORM	PPOS	
	BOS	BOS	**Padding**
	BOS	BOS	
1	Battle	NN	
2	-	HYPH	
3	tested	NN	
...	...	...	
17	the	DT	
18	first	JJ	
19	of	IN	
20	their	PRP$	
21	countrymen	NNS	**Input features**
22	to	TO	
23	**visit**	VB	**Predicted tag**
24	Mexico		↓
25	,		
26	a		
27	boatload		
...	...	...	
34	years		
35	ago		
36	.		
	EOS		**Padding**
	EOS		



From Feed-Forward to Recurrent (II)



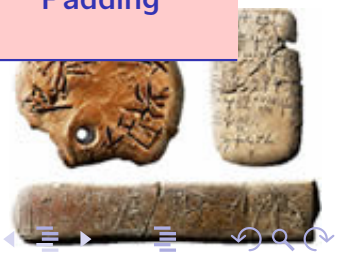
Input: Embeddings (GloVe in the examples)

Output: Softmax to predict the parts of speech

Architecture 3: Recurrent Structure

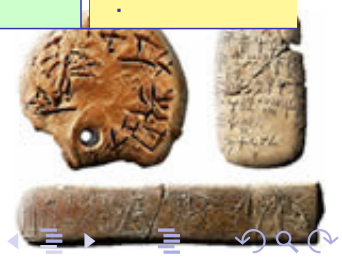
- In 2000, the CoNLL winners used *dynamic tags*
- The feature vector incorporates past predictions:
  $(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1})$,
  to predict the part-of-speech tag t_i at index i.
- Here:
  (countrymen, to, visit, Mexico, ",", NNS, TO)
  to predict VB

ID	FORM	PPOS	
	BOS	BOS	**Padding**
	BOS	BOS	
1	Battle	NN	
2	-	HYPH	
3	tested	NN	
...	...	...	
17	the	DT	
18	first	JJ	
19	of	IN	
20	their	PRP$	
21	countrymen	NNS	**Input features**
22	to	TO	
23	**visit**	VB	**Predicted tag**
24	Mexico		↓
25	,		
26	a		
27	boatload		
...	...	...	
34	years		
35	ago		
36	.		
	EOS		**Padding**
	EOS		

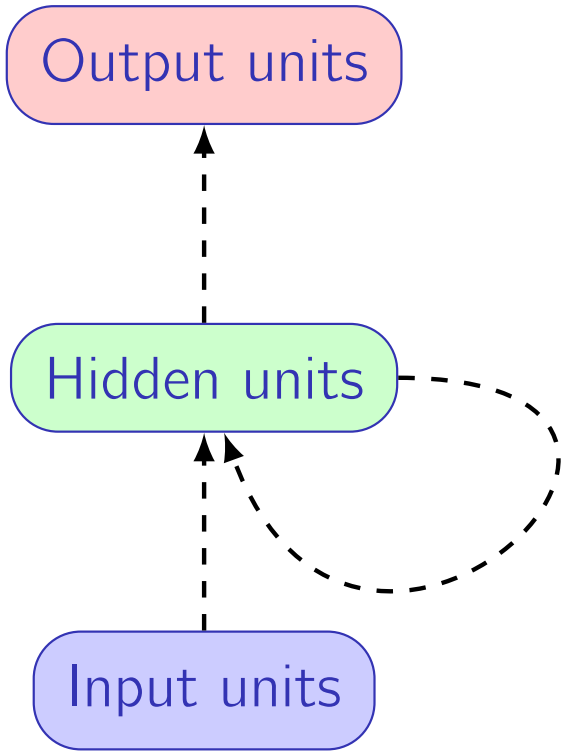


Feature Vectors

ID	Feature vectors: X							PPOS: ŷ
	w_{i-2}	w_{i-1}	w_i	w_{i+1}	w_{i+2}	t_{i-2}	t_{i-1}	
1	BOS	BOS	Battle	-	tested	BOS	BOS	NN
2	BOS	Battle	-	tested	Japanese	BOS	NN	HYPH
3	Battle	-	tested	Japanese	industrial	NN	HYPH	JJ
...	...	...	...	...	...	...	...	...
19	the	first	of	their	countrymen	DT	JJ	IN
20	first	of	their	countrymen	to	JJ	IN	PRP$
21	of	their	countrymen	to	visit	IN	PRP$	NNS
22	their	countrymen	to	visit	Mexico	PRP$	NNS	TO
23	countrymen	to	visit	Mexico	,	NNS	TO	VB
24	to	visit	Mexico	,	a	TO	VB	NNP
25	visit	Mexico	,	a	boatload	VB	NNP	,
...	...	...	...	...	...	...	...	...
34	ashore	375	years	ago	.	RB	CD	NNS
35	375	years	ago	.	EOS	CD	NNS	RB
36	years	ago	.	EOS	EOS	NNS	RB	.

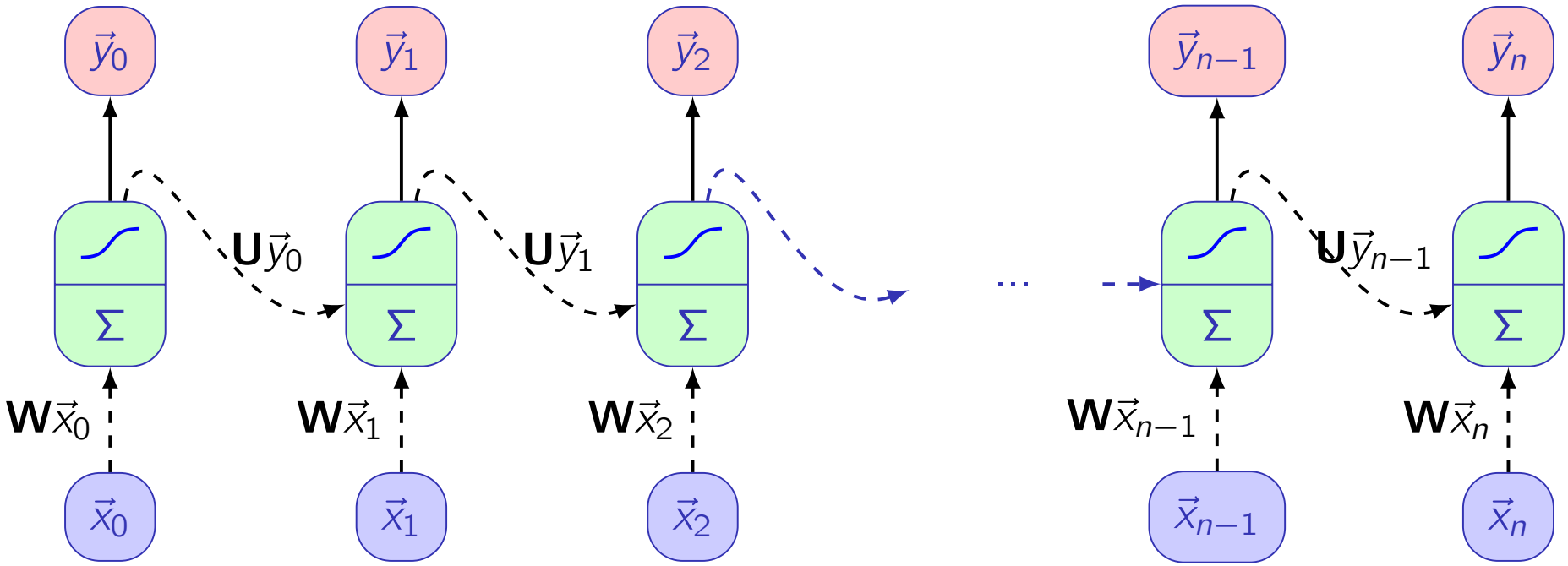


The RNN Architecture



A simple recurrent neural network; the dashed lines represent trainable connections.

The RNN Architecture (Unfolded)



The network unfolded in time. Equation used by implementations[1].

$$\mathbf{y}_{(t)} = \tanh(W\mathbf{x}_{(t)} + U\mathbf{y}_{(t-1)} + \mathbf{b})$$

[1] See: https://pytorch.org/docs/stable/nn.html#torch.nn.RNN

Input Format for RNNs

The input format is different from feed forward networks.
We need to build two lists: one for the input and the other for the output

y	DET	NOUN	VERB	DET	NOUN

x	The	waiter	brought	the	meal

All the vectors in a same batch must have the same length. We pad them:

y	PAD	PAD	PAD	DET	NOUN	VERB	DET	NOUN

x	PAD	PAD	PAD	The	waiter	brought	the	meal

We could apply the padding after

Building the Sequences

```
def build_sequences(corpus_dict, key_x='form', key_y='pos',
                    tolower=True):
    X, Y = [], []
    for sentence in corpus_dict:
        x, y = [], []
        for word in sentence:
            x += [word[key_x]]
            y += [word[key_y]]
        if tolower:
            x = list(map(str.lower, x))
        X += [x]
        Y += [y]
    return X, Y
```

At this point, we have **x** and **y** vectors of symbols

Building Index Sequences

0 is for the padding symbol and 1 for the unknown words

```
idx_word = dict(enumerate(vocabulary_words, start=2))
idx_pos = dict(enumerate(pos, start=2))
word_idx = {v: k for k, v in idx_word.items()}
pos_idx = {v: k for k, v in idx_pos.items()}
```

At this point, we have **x** and **y** vectors of numbers

Padding the Index Sequences

We build the complete X_idx and Y_idx matrices for the whole corpus
And we pad the matrices:

```
X_train_padded = pad_sequence(X_train_idx, batch_first=True)
Y_train_padded = pad_sequence(Y_train_idx, batch_first=True)

X_val_padded = pad_sequence(X_val_idx, batch_first=True)
Y_val_padded = pad_sequence(Y_val_idx, batch_first=True)
```

See: `https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pad_sequence.html`
`pad_sequences` can have an argument that specifies the padding value `padding_value`
The padded sentences must have the same length in a batch. This is automatically computed by PyTorch

Batch First

Batch-first ordering with these segments:

Sing, O goddess, ||the anger || of Achilles son of Peleus, || that brought countless ills || upon the Achaeans.

$$X = \begin{bmatrix} \text{sing} & \text{o} & \text{goddess} & \text{PAD} & \text{PAD} \\ \text{the} & \text{anger} & \text{PAD} & \text{PAD} & \text{PAD} \\ \text{of} & \text{achilles} & \text{son} & \text{of} & \text{peleus} \\ \text{that} & \text{brought} & \text{countless} & \text{ills} & \text{PAD} \\ \text{upon} & \text{the} & \text{achaeans} & \text{PAD} & \text{PAD} \end{bmatrix}$$

PyTorch uses an optimized tensor ordering:

$$X = \begin{bmatrix} \text{sing} & \text{the} & \text{of} & \text{that} & \text{upon} \\ \text{o} & \text{anger} & \text{achilles} & \text{brought} & \text{the} \\ \text{goddess} & \text{PAD} & \text{son} & \text{countless} & \text{achaeans} \\ \text{PAD} & \text{PAD} & \text{of} & \text{ills} & \text{PAD} \\ \text{PAD} & \text{PAD} & \text{peleus} & \text{PAD} & \text{PAD} \end{bmatrix}$$

To use the batch-first convention, you have to set `batch_first=True`.

Recurrent Neural Networks (RNN)

```
def __init__(self, embedding_matrix, rnn_units, nbr_classes,
      freeze_embs=True, num_layers=1):
    super().__init__()
    embedding_dim = embedding_matrix.size()[-1]
    self.embeddings = nn.Embedding.from_pretrained(
      embedding_matrix, freeze=freeze_embs, padding_idx=0)
    self.dropout = nn.Dropout(DROPOUT)
    self.rnn = nn.RNN(embedding_dim, rnn_units,
                      num_layers=num_layers, dropout=DROPOUT,
                      batch_first=True)

    self.fc = nn.Linear(rnn_units, nbr_classes)
```

Recurrent Neural Networks (RNN)

```
def forward(self, sentence):
    embeds = self.embeddings(sentence)
    embeds = self.dropout(embeds)
    rnn_out, _ = self.rnn(embeds)
    rnn_out = F.relu(lstm_out)
    rnn_out = self.dropout(rnn_out)
    logits = self.fc(rnn_out)
    return logits
```

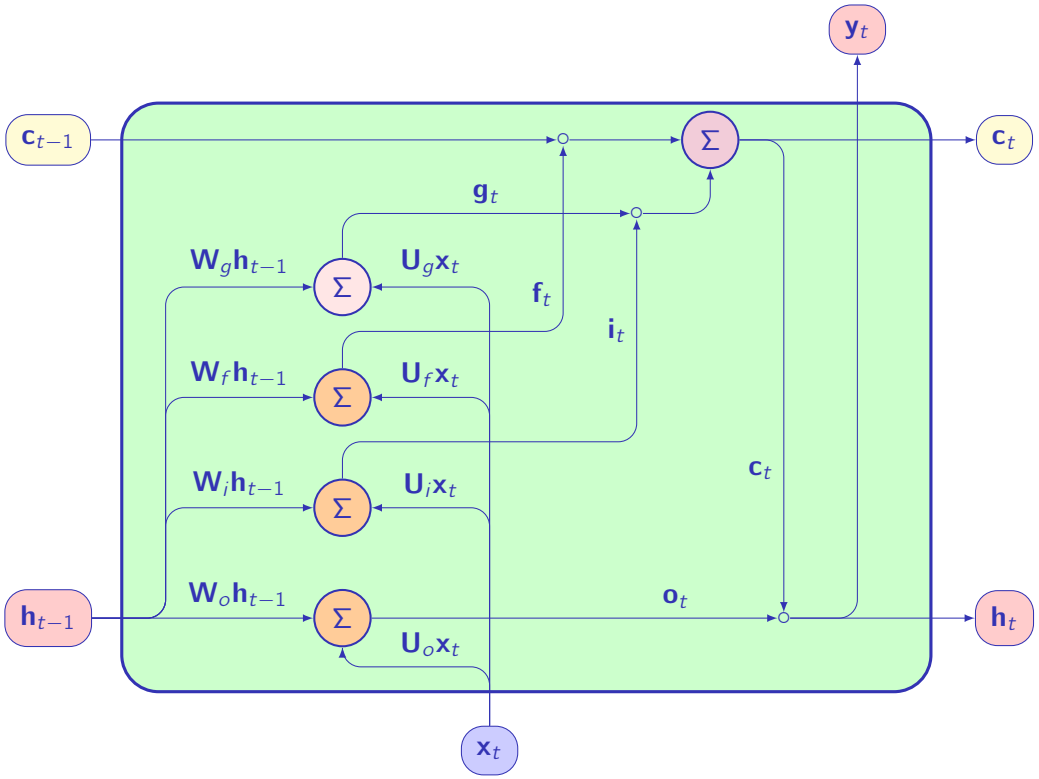
LSTMs

Simple RNNs use the previous output as input. They have then a very limited feature context.
Long short-term memory units (LSTM) are an extension to RNNs that can remember, possibly forget, information from longer or more distant sequences.
Given an input at index t, $\mathbf{x}_t$, a LSTM unit produces:

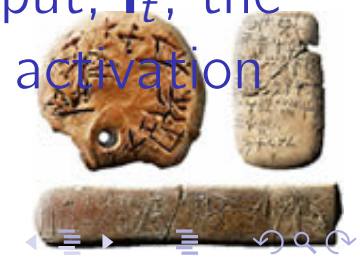
- A short term state, called $\mathbf{h}_t$ and
- A long-term state, called $\mathbf{c}_t$ or memory cell.

The short-term state, $\mathbf{h}_t$, is the unit output, i.e. $\mathbf{y}_t$; but both the long-term and short-term states are reused as inputs to the next unit.

The LSTM Architecture



An LSTM unit showing the data flow, where $\mathbf{g}_t$ is the unit input, $\mathbf{i}_t$, the input gate, $\mathbf{f}_t$, the forget gate, and $\mathbf{o}_t$, the output gate. The activation functions have been omitted



LSTM Equations

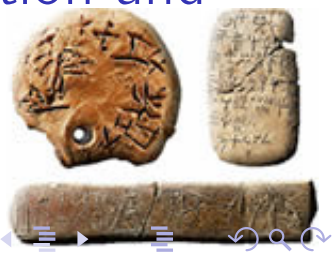
A LSTM unit starts from a core equation that is identical to that of a RNN:

$$\mathbf{g}_t = \tanh(W_g \mathbf{x}_t + U_g \mathbf{h}_{t-1} + \mathbf{b}_g).$$

From the previous output and current input, we compute three kinds of filters, or gates, that will control how much information is passed through the LSTM cell
The two first gates, **i** and **f**, defined as:

$$\begin{aligned} \mathbf{i}_t &= \text{activation}(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + \mathbf{b}_i), \\ \mathbf{f}_t &= \text{activation}(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + \mathbf{b}_f), \end{aligned}$$

model respectively how much we will keep from the base equation and how much we will forget from the long-term state.



LSTM Equations (II)

To implement this selective memory, we apply the two gates to the base equation and to the previous long-term state with the element-wise product (Hadamard product), denoted $\circ$, and we sum the resulting terms to get the current long-term state:

$$\mathbf{c}_t = \mathbf{i}_t \circ \mathbf{g}_t + \mathbf{f}_t \circ \mathbf{c}_{t-1}.$$

The third gate:

$$\mathbf{o}_t = \text{activation}(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + \mathbf{b}_o)$$

modulates the current long-term state to produce the output:

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t).$$

The LSTM parameters are determined by a gradient descent.
See also:
https://pytorch.org/docs/stable/nn.html#torch.nn.LSTM

Recurrent Neural Networks (RNN)

```
def __init__(self, embedding_matrix, lstm_units, nbr_classes,
    freeze_embs=True, num_layers=1, bidi_lstm=False):
  super().__init__()
  embedding_dim = embedding_matrix.size()[-1]
  self.embeddings = nn.Embedding.from_pretrained(
    embedding_matrix, freeze=freeze_embs, padding_idx=0)
  self.dropout = nn.Dropout(DROPOUT)
  self.lstm = nn.LSTM(embedding_dim, lstm_units,
              num_layers=num_layers, dropout=DROPOUT,
              batch_first=True, bidirectional=bidi_lstm)
  if not bidi_lstm:
      self.fc = nn.Linear(lstm_units, nbr_classes)
  else:
      # twice the units if bidirectional
      self.fc = nn.Linear(2*lstm_units, nbr_classes)
```

Recurrent Neural Networks (RNN)

```
def forward(self, sentence):
    embeds = self.embeddings(sentence)
    embeds = self.dropout(embeds)
    lstm_out, _ = self.lstm(embeds)
    lstm_out = F.relu(lstm_out)
    lstm_out = self.dropout(lstm_out)
    logits = self.fc(lstm_out)
    return logits
```

Code Example

Jupyter Notebook: `ch10-pos-tagger_lstm_embs.ipynb`

Segment Recognition

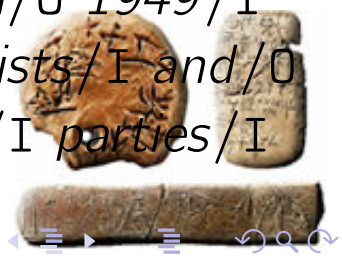
Group detection – chunking –:

Brackets: [$_{NG}$ The government $_{NG}$] has [$_{NG}$ other agencies and instruments $_{NG}$] for pursuing [$_{NG}$ these other objectives $_{NG}$] .

Tags: *The*/I *government*/I *has*/O *other*/I *agencies*/I *and*/I *instruments*/I *for*/O *pursuing*/O *these*/I *other*/I *objectives*/I ./O

Brackets: Even [$_{NG}$ Mao Tse-tung $_{NG}$] [$_{NG}$'s China $_{NG}$] began in [$_{NG}$ 1949 $_{NG}$] with [$_{NG}$ a partnership $_{NG}$] between [$_{NG}$ the communists $_{NG}$] and [$_{NG}$ a number $_{NG}$] of [$_{NG}$ smaller, non-communists parties $_{NG}$] .

Tags: *Even*/O *Mao*/I *Tse-tung*/I *'s*/B *China*/I *began*/O *in*/O *1949*/I *with*/O *a*/I *partnership*/I *between*/O *the*/I *communists*/I *and*/O *a*/I *number*/I *of*/O *smaller*/I *,*/I *non-communists*/I *parties*/I ./O



Segment Categorization

Tages extendible to any type of chunks: nominal, verbal, etc.
For the IOB scheme, this means tags such as I.Type, O.Type, and B.Type, Types being NG, VG, PG, etc.
In CoNLL 2000, ten types of chunks

Word	POS	Group	Word	POS	Group
He	PRP	B-NP	*to*	TO	B-PP
reckons	VBZ	B-VP	*only*	RB	B-NP
the	DT	B-NP	*£*	#	I-NP
current	JJ	I-NP	*1.8*	CD	I-NP
account	NN	I-NP	*billion*	CD	I-NP
deficit	NN	I-NP	*in*	IN	B-PP
will	MD	B-VP	*September*	NNP	B-NP
narrow	VB	I-VP	.	.	O

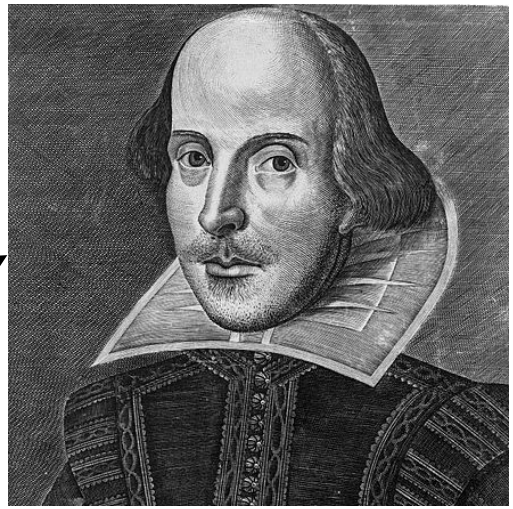
Noun groups (NP) are in red and verb groups (VP) are in blue.

IOB Annotation for Named Entities

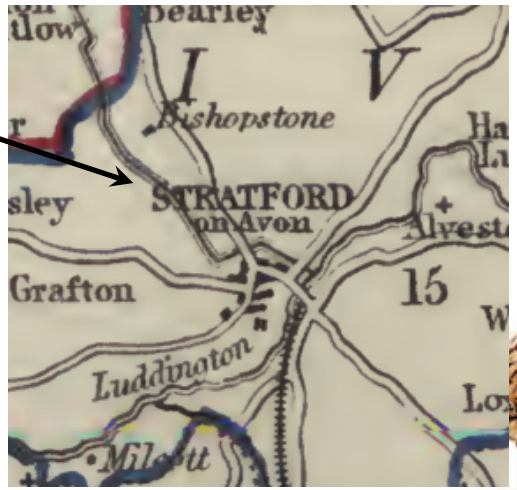
CoNLL 2002	
Words	**Named entities**
Wolff	B-PER
,	O
currently	O
a	O
journalist	O
in	O
Argentina	B-LOC
,	O
played	O
with	O
Del	B-PER
Bosque	I-PER
in	O
the	O
final	O
years	O
of	O
the	O
seventies	O
in	O
Real	B-ORG
Madrid	I-ORG
.	O

CoNLL 2003			
Words	**POS**	**Groups**	**Named entities**
U.N.	NNP	I-NP	I-ORG
official	NN	I-NP	O
Ekeus	NNP	I-NP	I-PER
heads	VBZ	I-VP	O
for	IN	I-PP	O
Baghdad	NNP	I-NP	I-LOC
.	.	O	O

Named Entities: Proper Nouns



William Shakespeare was born and brought up in Stratford-upon-Avon



Others Entities: Common Nouns

Meeting with our guest on the landing at lunchtime

Evaluation

There are different kinds of measures to evaluate the performance of machine learning techniques, for instance:

- Precision and recall in information retrieval and natural language processing;
- The *receiver operating characteristic* (ROC) in medicine.

	Positive examples: P	**Negative examples:** N
Classified as P	True positives: A	False positives: B
Classified as N	False negatives: C	True negatives: D

More on the receiver operating characteristic here: `http://en.wikipedia.org/wiki/Receiver_operating_characteristic`

Recall, Precision, and the F-Measure

The **accuracy** is $\frac{|A \cup D|}{|P \cup N|}$.

Recall measures how much relevant examples the system has classified correctly, for P:

$$\text{Recall} = \frac{|A|}{|A \cup C|}.$$

Precision is the accuracy of what has been returned, for P:

$$\text{Precision} = \frac{|A|}{|A \cup B|}.$$

Recall and precision are combined into the **F-measure**, which is defined as the harmonic mean of both numbers:

$$F = \frac{2 \cdot \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Evaluation: Accuracy, precision, and recall

For noun groups with the predicted output:

	Word	POS	Group	Predicted			Word	POS	Group	Predicted	
	He	PRP	B-NP	B-NP			*to*	TO	B-PP	B-PP	
	reckons	VBZ	B-VP	B-VP			*only*	RB	B-NP	B-NP	X
	the	DT	B-NP	B-NP	X		*£*	#	I-NP	I-NP	X
	current	JJ	I-NP	**B-NP**	X		*1.8*	CD	I-NP	**B-NP**	X
	account	NN	I-NP	I-NP	X		*billion*	CD	I-NP	I-NP	X
	deficit	NN	I-NP	I-NP	X		*in*	IN	B-PP	B-PP	
	will	MD	B-VP	B-VP			*September*	NNP	B-NP	B-NP	
	narrow	VB	I-VP	I-VP			*.*	.	O	O	

There are 16 chunk tags, 14 are correct: Accuracy = $\frac{14}{16} = 0.875$

There are 4 noun groups, the system retrieved 2 of them: Recall = $\frac{2}{4} = 0.5$

The system identified 6 noun groups, two are correct: Precision = $\frac{2}{6} = 0.33$

Harmonic mean = $2 \times \frac{0.33 \times 0.5}{0.33 + 0.5} = 0.4$