Language Technology

Chapter 14: Part-of-Speech and Sequence Annotation

Pierre Nugues

Pierre.Nugues@cs.lth.se

October 17, 2024

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 part-of-speech tagging, chunking, and named entity recognition.

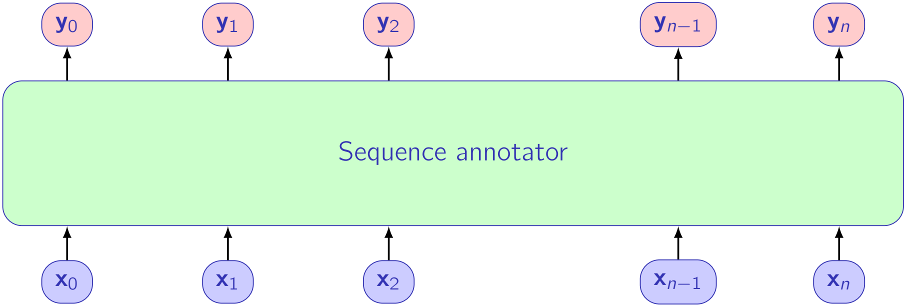
Model

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

Sequence Annotator

Sequence annotation:



Designing a Part-of-Speech Tagger

We will now create part-of-speech taggers

No unique solution

We will examine four architectures:

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

Annotated Corpora

- The Penn Treebank is one of the first very popular annotated corpus
- The Universal dependencies

Demo: `https://universaldependencies.org/`

Training Set (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

Training Set

Part-of-speech taggers use a training set where every word is hand-annotated (EWT).

ID	FORM	LEMMA	UPOS	FEATS
1	Or	or	CCONJ	_
2	you	you	PRON	Case=Nom|Person=2|PronType=Prs
3	can	can	AUX	VerbForm=Fin
4	visit	visit	VERB	VerbForm=Inf
5	temples	temple	NOUN	Number=Plur
6	or	or	CCONJ	_
7	shrines	shrine	NOUN	Number=Plur
8	in	in	ADP	_
9	Okinawa	Okinawa	PROPN	Number=Sing
10	.	.	PUNCT	_

UPOS

Words	Possible tags	Example of use	UPOS
that	Subordinating conjunction	*That he can swim is good*	SCONJ
	Determiner	*That white table*	DET
	Adverb	*It is not that easy*	ADV
	Pronoun	*That is the table*	PRON
	Relative pronoun	*The table that collapsed*	PRON
round	Verb	*Round up the usual suspects*	VERB
	Preposition	*Turn round the corner*	ADP
	Noun	*A big round*	NOUN
	Adjective	*A round box*	ADJ
	Adverb	*He went round*	ADV
table	Noun	*That white table*	NOUN
	Verb	*I table that*	VERB
might	Noun	*The might of the wind*	NOUN
	Modal verb	*She might come*	AUX
collapse	Noun	*The collapse of the empire*	NOUN
	Verb	*The empire can collapse*	VERB

Baseline

You just count the parts of speech in the annotated corpus

Words	Parts-of-speech counts	Most frequent POS	Correct POS
That	PRON: 58, DET: 15, SCONJ: 6	PRON	DET
round	NOUN: 4, ADV: 3, ADJ: 2, ADP: 2	NOUN	ADJ
table	NOUN: 14	NOUN	NOUN
might	AUX: 77	AUX	AUX
collapse	NOUN: 2, VERB: 1	NOUN	VERB

Accuracy: 86.4% on the English Web Treebank (EWT)

Confusion Matrix

↓**Correct**	**Tagger** →									
	ADJ	ADP	ADV	AUX	CCONJ	DET	NOUN	PRON	PROPN	SCONJ	VERB
ADJ	82.8	0.6	1.5	0.	0.	0.1	1.9	0.	11.5	0.	1.6
ADP	0.	88.2	0.4	0.	0.	0.	0.	0.	0.3	0.6	0.
ADV	5.3	7.1	78.5	0.1	0.2	1.3	0.8	2.5	2.1	1.4	0.1
AUX	0.	0.	0.	88.9	0.	0.	0.1	0.1	0.3	0.	6.5
CCONJ	0.	0.1	0.	0.	99.7	0.	0.	0.	0.1	0.	0.
DET	0.2	0.	0.1	0.	0.2	96.8	0.1	1.7	0.1	0.9	0.
NOUN	0.8	0.1	0.2	0.1	0.	0.	76.2	0.	19.	0.1	3.1
PRON	0.	0.	0.	0.	0.	2.1	0.	93.2	0.1	4.4	0.
PROPN	1.1	0.3	0.	0.1	0.	0.	4.1	0.	93.6	0.	0.4
SCONJ	0.	33.3	1.6	0.	0.	0.	0.	0.3	1.6	60.4	0.
VERB	0.6	0.9	0.2	3.6	0.	0.	5.7	0.	7.2	0.	81.5

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.

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	UPOS	
		BOS		**Padding**
		BOS		
1		Or	CCONJ	
2		you	PRON	
3		can	AUX	
4	**Input features**	**visit**	VERB	**Predicted tag**
4		temples		↓
6		or		
7		shrines		
8		in		
9		Okinawa		
10		.		
		EOS		**Padding**
		EOS		

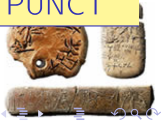
Feed-Forward Structure

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:
(you, can, visit, temples, or)
to predict VERB

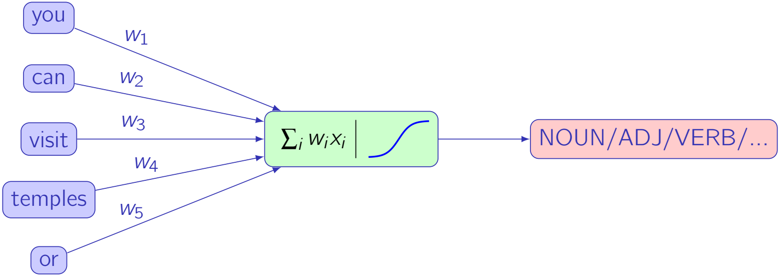
ID		FORM	UPOS	
		BOS		**Padding**
		BOS		
1		Or	CCONJ	
2		you	PRON	
3		can	AUX	
4	**Input features**	**visit**	VERB	**Predicted tag**
4		temples		↓
6		or		
7		shrines		
8		in		
9		Okinawa		
10		.		
		EOS		**Padding**
		EOS		

Feature Vectors

ID	**Feature vectors: X**					**UPOS: y**
	w_{i-2}	w_{i-1}	w_i	w_{i+1}	w_{i+2}	
1	BOS	BOS	Or	you	can	CCONJ
2	BOS	Or	you	can	visit	PRON
3	Or	you	can	visit	temples	AUX
4	you	can	visit	temples	or	VERB
5	can	visit	temples	or	shrines	NOUN
6	visit	temples	or	shrines	in	CCONJ
7	temples	or	shrines	in	Okinawa	NOUN
8	or	shrines	in	Okinawa	.	ADP
9	shrines	in	Okinawa	.	EOS	PROPN
10	in	Okinawa	.	EOS	EOS	PUNCT



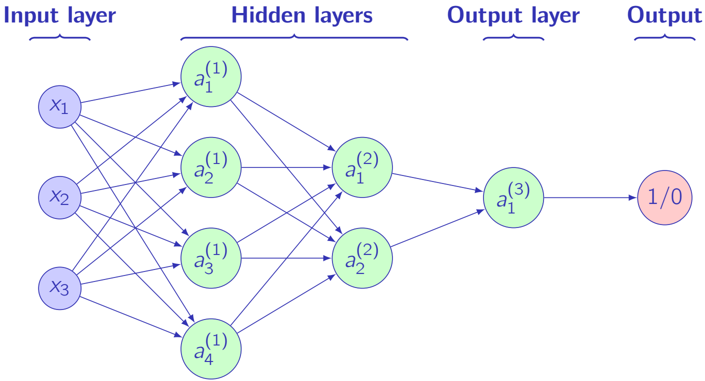
Feed Forward (Multinomial) (II)



Input: one-hot encoding

Output: Softmax to predict the parts of speech

Feed Forward (Multilayer)



For the first layer, we have:

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

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)
```

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
X_cat = [{0: ’__BOS__’, 1: ’__BOS__’, 2: ’Or’, 3: ’you’, 4: ’c
 {0: ’__BOS__’, 1: ’Or’, 2: ’you’, 3: ’can’, 4: ’visit’},
 {0: ’Or’, 1: ’you’, 2: ’can’, 3: ’visit’, 4: ’temples’},
 ...
 {0: ’or’, 1: ’shrines’, 2: ’in’, 3: ’Okinawa’, 4: ’.’},
 {0: ’shrines’, 1: ’in’, 2: ’Okinawa’, 3: ’.’, 4: ’__EOS__’},
 {0: ’in’, 1: ’Okinawa’, 2: ’.’, 3: ’__EOS__’, 4: ’__EOS__’}]
 v = DictVectorizer(sparse=False)
X = v.fit_transform(X_cat)
```

Code Example

Logistic regression with sklearn

Jupyter Notebook: https://github.com/pnugues/pnlp/blob/main/notebooks/14_02_pos_lr.ipynb

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))
```

Code Example

Jupyter Notebook: https://github.com/pnugues/pnlp/blob/main/notebooks/14_03_pos_ff.ipynb

Architecture 2: 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

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

Embeddings

Not a built-in feature of sklearn

PyTorch:

```
model = nn.Sequential(
    nn.Embedding.from_pretrained(
        embedding_table, freeze=False),
    nn.Flatten(),
    nn.Linear(5 * embedding_table.size(dim=1),
              len(pos2idx))
)
```

Code Example

Jupyter Notebook: `https://github.com/pnugues/pnlp/blob/main/notebooks/14_04_pos_ff_embs.ipynb`

Architecture 3: Recurrent Neural Networks

In feed-forward networks, predictions in a sequence of classifications are independent.

In many cases, given an input, the prediction also depends on the previous decision.

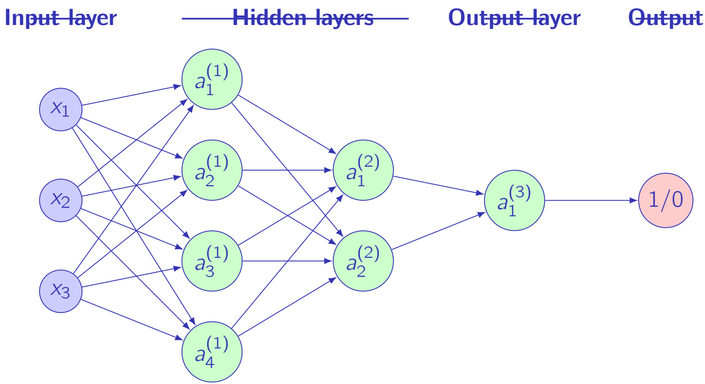
For instance, in weather forecast, if the input is the temperature and the output is rain/not rain, for a same temperature, it the previous output was rain, the next one is likely to be rain.

Recurrent neural networks (RNN) try to model these dependencies

In this lecture, we will examine recurrent neural networks for:

1. Categorization, i.e. given a sentence, predict its category, one output
2. Sequence annotation, i.e. given a sentence, predict a sequence of symbols, a sequence of outputs

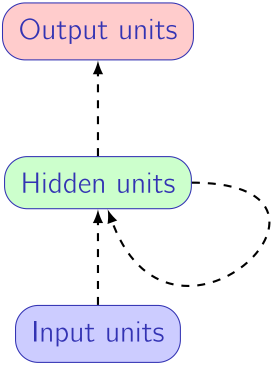
Feed Forward (Reminder)



For the first layer, we have:

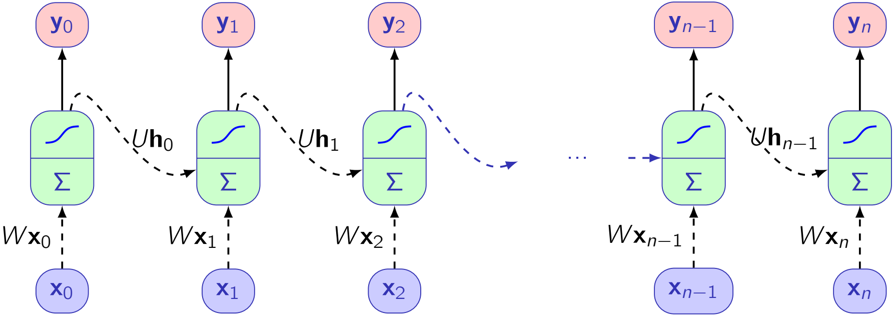
$$\text{activation}(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}).$$

The RNN Architecture



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

The Unfolded RNN Architecture



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

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

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

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 apply the padding after in 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`

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

PyTorch

```
class Model(nn.Module):

    def __init__(self, embedding_table, hidden_size,
                 nbr_classes, freeze_embs=True,
                 num_layers=1, bidirectional=False):
        super().__init__()
        embedding_dim = embedding_table.size(dim=-1)
        self.embeddings = nn.Embedding.from_pretrained(
            embedding_table,
            freeze=freeze_embs,
            padding_idx=0)
        self.recurrent = nn.RNN(embedding_dim,
                                hidden_size,
                                batch_first=True,
                                num_layers=num_layers,
                                bidirectional=bidirectional)
```

PyTorch

```
class Model(nn.Module):

    def __init__(self, embedding_table, hidden_size,
...
        if not bidirectional:
            self.fc = nn.Linear(hidden_size, nbr_classes)
        else:
            # twice the units if bidirectional
            self.fc = nn.Linear(2*hidden_size, nbr_classes)
```

PyTorch

```
class Model(nn.Module):

...
    def forward(self, sentence):
        embeds = self.embeddings(sentence)
        rec_out, _ = self.recurrent(embeds)
        logits = self.fc(rec_out)
        return logits
```

Code Example

Jupyter Notebook: https://github.com/pnugues/pnlp/blob/main/notebooks/14_05_pos_lstm.ipynb

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.

We use the short-term state, $\mathbf{h}_t$, to produce the output, i.e. $\mathbf{y}_t$ with a linear layer and a softmax activation; but both the long-term and short-term states are reused as inputs to the next unit.

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:

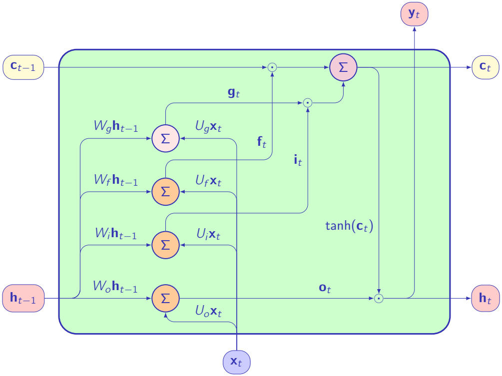
$$\mathbf{o}_t = \text{activation}(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + \mathbf{b}_o)$$

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/generated/torch.nn.LSTM.html

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

Building a LSTM with PyTorch

```
def __init__(self, lstm_units, nbr_classes, num_layers=1,
           bidi_lstm=False):
    super().__init__()
    self.dropout = nn.Dropout(DROPOUT)
    self.lstm = nn.LSTM(MAX_TOKENS + 2, 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)
```

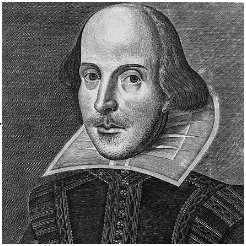
Building a LSTM with PyTorch

```
def forward(self, sentence):
        embeds = F.one_hot(sentence,
                num_classes=MAX_TOKENS + 2).float()
        lstm_out, (h_n, c_n) = self.lstm(embeds)
        lstm_last = F.relu(h_n[-1])
        lstm_last = self.dropout(lstm_last)
        logits = self.fc(lstm_last)
        return logits
```

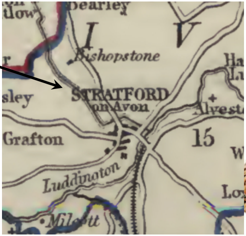
Recurrent Networks for Classification

- We can use a recurrent network to classify texts
- The IMDB dataset of movie reviews annotated as positive or negative
- In a feed-forward network, we build a representation of the documents using the vector space model or a dense representation (SBERT)
- In a recurrent architecture, the words will go through the network and we will use the last output to classify a text
- As vectorization, we can use a one-hot encoding or a dense representation of the words (GloVe)

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

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

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 $\dfrac{|A \cup D|}{|P \cup N|}$.

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	
He	PRP	B-NP	B-NP	
reckons	VBZ	B-VP	B-VP	
the	DT	B-NP	B-NP	X
current	JJ	I-NP	**B-NP**	X
account	NN	I-NP	I-NP	X
deficit	NN	I-NP	I-NP	X
will	MD	B-VP	B-VP	
narrow	VB	I-VP	I-VP	
to	TO	B-PP	B-PP	
only	RB	B-NP	B-NP	X
£	#	I-NP	I-NP	X
1.8	CD	I-NP	**B-NP**	X
billion	CD	I-NP	I-NP	X
in	IN	B-PP	B-PP	
September	NNP	B-NP	B-NP	
.	.	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 $= 2 \times \frac{0.33 \times 0.5}{0.33 + 0.5} = 0.4$

Tokenization Revisited

- Some Asian languages do not include tokenization marks as in:
  然而，這樣的處理也衍生了一些問題。
  'However, this treatment also created some problems.'
  From Universaldependencies.org
- Tokenized as: 然而||，||這樣||的||處理||也||衍生||了||一些||問題||。
- Shao proposed the tokenization with the tagset: B, I, E, and S, where
  - B is the beginning of a word, I is inside, and E is the end.
  - S is for a single-character word.

然	而	，	這	樣	的	處	理	也	衍	生	了	一	些	問	題	。
B	E	S	B	E	S	B	E	S	B	E	S	B	E	B	E	S

Adaptation to Other Languages

In other languages, we have tokenization markers, mostly spaces.
We mark them with the X tag.
An example in French:

Chars:	`On considère qu'environ 50 000 Allemands du Wartheland ont péri pendant la période.`
Tags:	`BEXBIIIIIIIEXBIEBIIIIIEXBIIIIEXBIIIIIIIEXBEXBIIIIIIIIEXBIEXBIIEXBIIIIIEXBEXBIIIIIES`

Finally, we can use a final tag T to mark the end of a sentence.
This will enable us to carry out jointly tokenization and the sentence segmentation of a text.

Training the Model

The sentence # sent_id = test-s1
text = 然而，這樣的處理也衍生了一些問題。
The tokenized version from universal dependencies:

ID	FORM	LEMMA	UPOS	XPOS	FEATS	HEAD	DEPREL	DEPS	MISC
1	然而	然而	ADV	RB	_	7	mark	_	SpaceAfter=No
2	，	，	PUNCT	,	_	7	punct	_	SpaceAfter=No
3	這樣	這樣	PRON	PRD	_	5	det	_	SpaceAfter=No
4	的	的	PART	DEC	Case=Gen	3	case	_	SpaceAfter=No
5	處理	處理	NOUN	NN	_	7	nsubj	_	SpaceAfter=No
6	也	也	ADV	RB	_	7	mark	_	SpaceAfter=No
7	衍生	衍生	VERB	VV	_	0	root	_	SpaceAfter=No
8	了	了	AUX	AS	Aspect=Perf	7	aux	_	SpaceAfter=No
9	一些	一些	ADJ	JJ	_	10	amod	_	SpaceAfter=No
10	問題	問題	NOUN	NN	_	7	obj	_	SpaceAfter=No
11	。	。	PUNCT	.	_	7	punct	_	SpaceAfter=No