EDAN95
Applied Machine Learning
`http://cs.lth.se/edan95/`
Lecture 9: Encoders-Decoders and Generative Learning

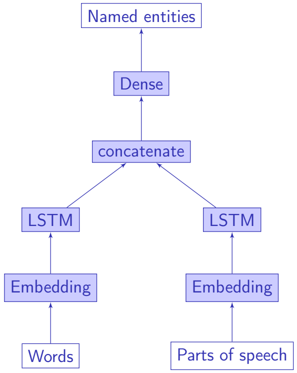
Pierre Nugues

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

December 3, 2018

The Functional Model

So far, we have used the `Sequential` model to build networks
These models correspond to pipelines with one input and one output



To build graphs, we need to use the functional model.

Comparing the Models

For a pipeline, the structure is nearly the same, with different classes:

- Sequential:

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu',
  input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

- Functional:

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

From Chollet, page 237

Building a Multi Input Model: Named Entity Recognition

CoNLL 2003

Words	PPOS	PGroups	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	O
Input	**Predicted by the organizers**		**Output**

1. The words are the input;
2. The CoNLL organizers have manually annotated the named entities and they are are the output;
3. The organizers have predicted the parts of speech and the groups to make the work easier for participants.

The Word Branch

We will now build a NER tagger that uses two inputs: the words and parts of speech
To build a multi input, we need the functional model and, at a certain point, merge the branches with `layers.concatenate()` function

```
text_vocabulary_size = len(word_set) + 2
text_input = Input(shape=(None,), dtype='int32', name='text')
embedded_text = layers.Embedding(text_vocabulary_size,
                      64, mask_zero=True)(text_input)
encoded_text = layers.LSTM(32,
                      return_sequences=True)(embedded_text)
```

The Part-of-Speech Branch

```
pos_vocabulary_size = len(pos_set) + 2
pos_input = Input(shape=(None,),
                       dtype='int32',
                       name='pos')
embedded_pos = layers.Embedding(pos_vocabulary_size,
                                32, mask_zero=True)(pos_input)
encoded_pos = layers.LSTM(16,
               return_sequences=True)(embedded_pos)
```

Merging and Common Part

```
concatenated = layers.concatenate(
            [encoded_text, encoded_pos],axis=-1)
ner_vocabulary_size = len(ner_set) + 2
ner = layers.Dense(ner_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, pos_input], ner)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

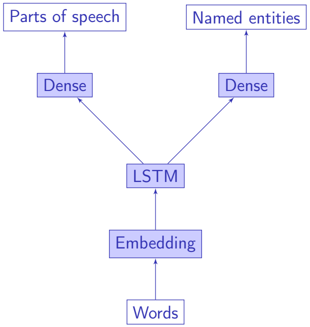
Code Example

The NER tagger with two inputs: the words and parts of speech and we will compare it to a sequential model

Jupyter Notebooks: `5.2-multiinput.ipynb` and `5.3-monoinput.ipynb`

Multiple Outputs

It is also possible to build a model with multiple outputs, for instance the word as input to predict the parts of speech and the named entities.



The Word Input

```
text_vocabulary_size = len(word_set) + 2
text_input = Input(shape=(None,), dtype='int32', name='text')
embedded_text = layers.Embedding(text_vocabulary_size,
                           64, mask_zero=True)(text_input)
encoded_text = layers.LSTM(32,
                     return_sequences=True)(embedded_text)
```

The POS output

```
pos_vocabulary_size = len(pos_set) + 2
pos_output = layers.Dense(pos_vocabulary_size,
                      activation='softmax',
                  name='pos')(encoded_text)
```

The NER Output

```
ner_vocabulary_size = len(ner_set) + 2
ner_output = layers.Dense(ner_vocabulary_size,
                          activation='softmax',
                          name='ner')(encoded_text)
```

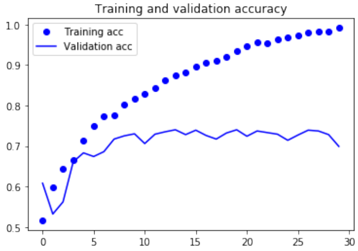
The Model

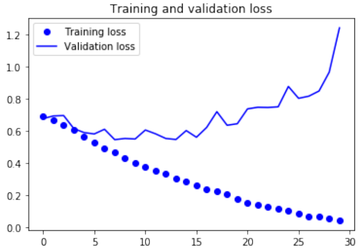
```
model = Model(text_input, [pos_output, ner_output])
model.compile(optimizer='rmsprop',
              loss=['categorical_crossentropy',
                    'categorical_crossentropy'],
              metrics=['acc'])

model.fit(X_words_idx,
          {'pos':Y_pos_idx_cat, 'ner':Y_ner_idx_cat},
          epochs=3, batch_size=128)
```

It is possible to build mode complex models, provided that they have the form of a directed acyclic graph. See the book.

Monitoring Training

We have seen different shapes of validation accuracies and loss:





15 epochs seem the optimal number and it is probably useless to run more. Keras provided callbacks for this.

Two Callbacks

1. keras.callbacks.EarlyStopping to stop training when validation scores do not improve;
2. keras.callbacks.ModelCheckpoint to save models

```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor='acc',
        patience=1,),
    keras.callbacks.ModelCheckpoint(
        filepath='my_model.h5',
        monitor='val_loss',
        save_best_only=True,)
 ]
```

From Chollet, page 250

Including the Callbacks

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
model.fit(x, y,
          epochs=10,
          batch_size=32,
          callbacks=callbacks_list,
          validation_data=(x_val, y_val))
```

You can also write your own callbacks, see Chollet, page 251-252

Tensorboard

Tensorboard is a visualization tool
You include it with a callback

```
callbacks = [
    keras.callbacks.TensorBoard(
        log_dir='tb_log_folder',
        histogram_freq=1
) ]
```

Demonstration

Tensorboard

Generative Learning

Words and characters have specific contexts of use.
Pairs of words like *strong* and *tea* or *powerful* and *computer* are not random associations.
Psychological linguistics tells us that it is difficult to make a difference between *writer* and *rider* without context
A listener will discard the improbable *rider of books* and prefer *writer of books*
A language model is the statistical estimate of a word sequence.
Originally developed for speech recognition
The language model component enables to predict the next word given a sequence of previous words

N-Grams

The types are the distinct words of a text while the tokens are all the words or symbols.
The phrases from *Nineteen Eighty-Four*

> *War is peace*
> *Freedom is slavery*
> *Ignorance is strength*

have 9 tokens and 7 types.
Unigrams are single words
Bigrams are sequences of two words
Trigrams are sequences of three words

Trigrams

Word	Rank	More likely alternatives
We	9	*The This One Two A Three Please In*
need	7	*are will the would also do*
to	1	
resolve	85	*have know do...*
all	9	*the this these problems...*
of	2	*the*
the	1	
important	657	*document question first...*
issues	14	*thing point to...*
within	74	*to of and in that...*
the	1	
next	2	*company*
two	5	*page exhibit meeting day*
days	5	*weeks years pages months*

Language Models and Generation

Using a n-gram language model, we can generate a sequence of words.
Starting from a first word, w_1, we extract the conditional probabilities: $P(w_2|w_1)$.
We could take the highest value, but it would always generate the same sequence.
Instead, we will draw our words from a multinomial distribution using `np.random.multinomial()`.
Given a probability distribution, this function draws a sample that complies the distribution.
Having, $P(want|I) = 0.5$, $P(wish|I) = 0.3$, $P(will|I) = 0.2$, the function will draw wish 30% of the time.

Code Example

Generating sequences with Bayesian probabilities
Jupyter Notebooks: `5.7-generation.ipynb`

Generating Character Sequences with LSTMs

In the previous example, we used words. We can use characters instead.
We also used Bayesian probabilities. We can use LSTMs instead.
This is the idea of Chollet's program, pages 272-278.
$\mathbf{X}$ consists of sequences of 60 characters with a step of 3 characters
$\mathbf{y}$ is the character following the sequence
Let us use this excerpt:

is there not ground for suspecting that all philosophers

and 10 characters, where ␣ marks a space:

$$\mathbf{X} = \begin{bmatrix} \mathrm{i} & \mathrm{s} & \text{␣} & \mathrm{t} & \mathrm{h} & \mathrm{e} & \mathrm{r} & \mathrm{e} & \text{␣} & \mathrm{n} \\ \mathrm{t} & \mathrm{h} & \mathrm{e} & \mathrm{r} & \mathrm{e} & \text{␣} & \mathrm{n} & \mathrm{o} & \mathrm{t} & \text{␣} \\ \mathrm{r} & \mathrm{e} & \text{␣} & \mathrm{n} & \mathrm{o} & \mathrm{t} & \text{␣} & \mathrm{g} & \mathrm{r} & \mathrm{o} \\ \mathrm{n} & \mathrm{o} & \mathrm{t} & \text{␣} & \mathrm{g} & \mathrm{r} & \mathrm{o} & \mathrm{u} & \mathrm{n} & \mathrm{d} \\ \text{␣} & \mathrm{g} & \mathrm{r} & \mathrm{o} & \mathrm{u} & \mathrm{n} & \mathrm{d} & \text{␣} & \mathrm{f} & \mathrm{o} \end{bmatrix}; \mathbf{y} = \begin{bmatrix} \mathrm{o} \\ \mathrm{g} \\ \mathrm{u} \\ \text{␣} \\ \mathrm{r} \end{bmatrix}$$

Generating Character Sequences with LSTMs

In addition, Chollet uses a "temperature" function to transform the probability distribution: sharpen or damps it.

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

with the input `[0.2, 0.5, 0.3]`, we obtain:

- Temperature = 2, `[0.26275107 0.41544591 0.32180302]`
- Temperature = 1, `[0.2 0.5 0.3]`
- Temperature = 0.5 `[0.10526316 0.65789474 0.23684211]`
- Temperature = 0.2 `[0.00941176 0.91911765 0.07147059]`

Code Example

Form Chollet's github repository:
Jupyter Notebooks: `8.1-text-generation-with-lstm.ipynb`

Machine Translation

Process of translating automatically a text in a source language into a target language
Started after the 2nd world war to translate text from Russian to English
Early working systems from French to English in Canada
Renewed huge interest with the advent of the web: Google claims it has more than 500m users daily worldwide, with 103 languages.
Massive progress permitted by the neural networks

Corpora for Machine Translation

Initial ideas in machine translation: use bilingual dictionaries and formalize grammatical rules to transfer them from a source language to a target language.

Statistical machine translation:

1. use very large bilingual corpora;
2. align the sentences or phrases, and
3. given a sentence in the source language, find the matching sentence in the target language.

Pioneered at IBM on French and English with Bayesian statistics.

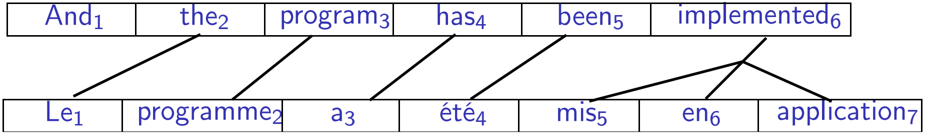
Neural nets are now dominant

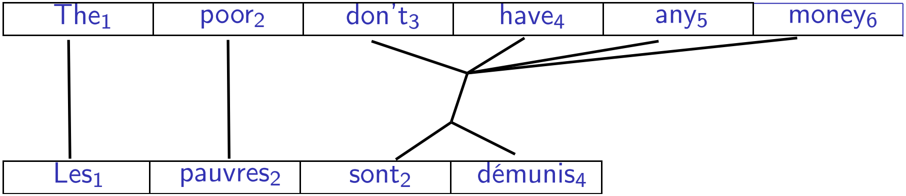
Parallel Corpora (Swiss Federal Law)

German	French	Italian
Art. 35 Milchtransport	**Art. 35 Transport du lait**	**Art. 35 Trasporto del latte**
1 Die Milch ist schonend und hygienisch in den Verarbeitungsbetrieb zu transportieren. Das Transportfahrzeug ist stets sauber zu halten. Zusammen mit der Milch dürfen keine Tiere und milchfremde Gegenstände transportiert werden, welche die Qualität der Milch beeinträchtigen können.	1 Le lait doit être transporté jusqu'à l'entreprise de transformation avec ménagement et conformément aux normes d'hygiène. Le véhicule de transport doit être toujours propre. Il ne doit transporter avec le lait aucun animal ou objet susceptible d'en altérer la qualité.	1 Il latte va trasportato verso l'azienda di trasformazione in modo accurato e igienico. Il veicolo adibito al trasporto va mantenuto pulito. Con il latte non possono essere trasportati animali e oggetti estranei, che potrebbero pregiudicarne la qualità.

Alignment (Brown et al. 1993)

Canadian Hansard





Translations with RNNs

RNN can easily map sequences to sequences, where we have two lists: one for the source and the other for the target

y	Le	serveur	apporta	le	plat
x	The	waiter	brought	the	meal

The **x** and **y** vectors must have the same length.
In our case, *a apporté* is more natural than *apporta* and it breaks the alignment, as well as in many other examples

Using the Hidden States

To solve the alignment problem, Sutskever al al. (2014) proposed (quoted from their paper, `https://arxiv.org/abs/1409.3215`):

- The simplest strategy for general sequence learning is to map the input sequence to a fixed-sized vector using one RNN, and then to map the vector to the target sequence with another RNN [...]
- it would be difficult to train the RNNs due to the resulting long term dependencies [...]. However, the Long Short-Term Memory (LSTM) is known to learn problems with long range temporal dependencies.
- LSTM estimate[s] the conditional probability $p(y_1, ..., y_{T'} | x_1, ..., x_T)$, where $(x_1, ..., x_T)$ is an input sequence and $y_1, ..., y_{T'}$ is its corresponding output sequence whose length T' may differ from T. The LSTM computes this conditional probability by first obtaining the fixed-dimensional representation v of the input sequence $(x1, ..., xT)$ given by the last hidden state of the LSTM, and then computing the probability of $y_1, ..., y_{T'}$ with a standard LSTM-LM formulation whose initial hidden state is set to the representation v of $x_1, ..., x_T$.

Sequence-to-Sequence Translation

We follow and reuse: `https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-html` from Chollet.

1. We start with input sequences from a domain (e.g. English sentences) and corresponding target sequences from another domain (e.g. French sentences).
2. An encoder LSTM turns input sequences to 2 state vectors (we keep the last LSTM state and discard the outputs).
3. A decoder LSTM is trained to turn the target sequences into the same sequence but offset by one timestep in the future, a training process called "teacher forcing" in this context. Is uses as initial state the state vectors from the encoder. Effectively, the decoder learns to generate `targets[t+1...]` given `targets[...t]`, conditioned on the input sequence.

Inference

Following Chollet, in inference mode, to decode unknown input sequences, we:

- Encode the input sequence into state vectors
- Start with a target sequence of size 1 (just the start-of-sequence character)
- Feed the state vectors and 1-char target sequence to the decoder to produce predictions for the next character
- Sample the next character using these predictions (we simply use argmax).
- Append the sampled character to the target sequence
- Repeat until we generate the end-of-sequence character or we hit the character limit.

Attention

For the latest developments, see: http://www.statmt.org/wmt18/
For a description of systems at Google, see https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html
For an example in Python, see, https://machinelearningmastery.com/encoder-decoder-attention-sequence-to-sequence-prediction-kera