université PARIS-SACLAY

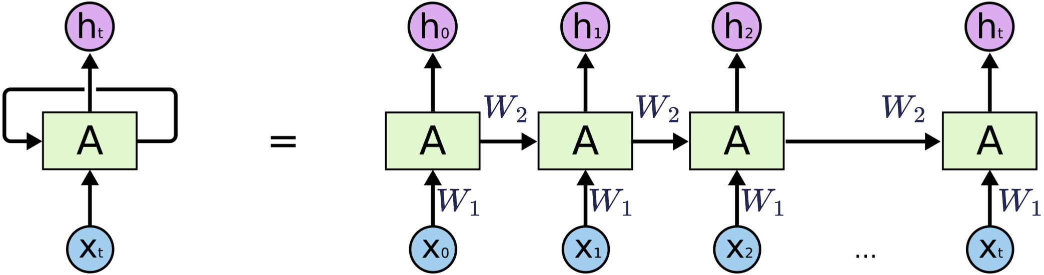
RÉSEAUX DE NEURONES RÉCURRENTS

Vincent Guigue,
inspiré des supports de Nicolas Baskiotis & Benjamin Piwowarski

MIA PARIS-SACLAY EKINOCS

AGROPARISTECH Institut des Sciences et Industries du Vivant et de l'Environnement

Recurrent Neural Network



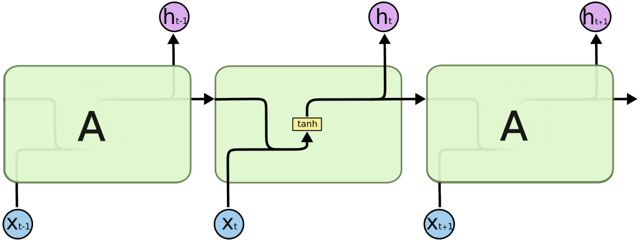
General RNN:

$$\mathbf{h}_t = \tanh(\mathbf{x}_t W_1 + \mathbf{h}_{t-1} W_2 + b) \text{ or } \tanh((x_t \oplus h_{t-1})W + b)$$

- **Few parameters**
- Ability to deals with **variable lengths**
- Capture the **dynamics** of the sequence

Cell details

Classical RNN :



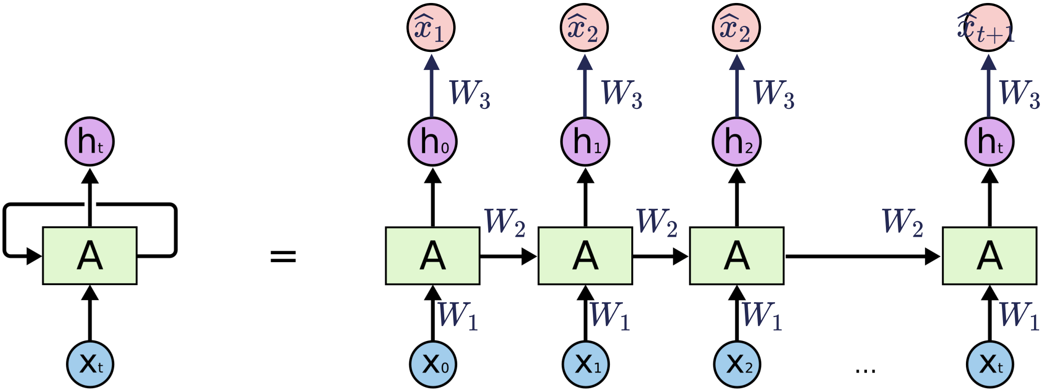
- Latent state $h_t \in \mathbb{R}^d$
- input $x_t \in \mathbb{R}^n$
- $h_t = f_W(h_{t-1}, x_t) = \tanh((x_t \oplus h_{t-1})W + b)$
- $W \in \mathbb{R}^{n \times d}$, $b \in \mathbb{R}$

  - Initial state: h_1
  - Sequence $[x_1, \ldots x_T] \Rightarrow [h_1, \ldots h_T]$
  - May be computed left $\rightarrow$ right (h_t) ... Or right $\rightarrow$ left (h_t^R)

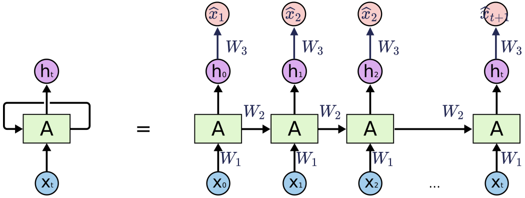
RNN... For which purpose?

- Predicting the next step (from h_T)
- Generation = prediction (in loop)
- Classifying a sequence (from h_T)
- Detecting an event (at every h_t)

Bi-RNN: use h_t & h_t^R
(h_T & h_1^R = caract. whole sequence)



Learn to generate



- Teacher forcing:

$$\hat{x}_{t+1} = g(h_t), \qquad h_{t+1} = f_W(h_t, x_{t+1})$$

- Free Generation

$$\hat{x}_{t+1} = g(h_t), \qquad h_{t+1} = f_W(h_t, \hat{x}_{t+1})$$

In practice:

- Start with teacher forcing (more stable) $\Rightarrow$ then switch to free generation
- Sequential learning also = more complicated solution

$\Rightarrow$ **Reinforcement Learning**

RNN In practice

```
# Define the model
model = torch.nn.RNN(input_size=100, hidden_size=50,
                     num_layers=1,
                     nonlinearity='tanh', bias=True)

# data (x_1,...,x_T)
# Seq. length T=23 & batch of size 10
# Input x_t \in R^n dimension 100
seq_x = torch.rand(23, 10, 100)

# seq_h is a tensor (23, 10, 50)
# For classification purpose, we use last_h
seq_h, last_h = model(seq_x)
```

Convention

Tensor of size **time x sample x representation**

Specificities & difficulties

Problems with RNN

Simple use case, linear + no entries

- $h_{t+1} = Wh_t$
- t step $\Rightarrow$ computing W^t
- Given the decomposition: $W = U\Sigma U^T$, $\Sigma =$ diag matrix of eigen values

Gradient computation leads to:

$$\frac{\partial E}{\partial h_t} = \frac{\partial E}{\partial h_N}(W^T)^{N-t} = \Delta U \begin{pmatrix} \sigma_1^{N-t} & & \\ & \ddots & \\ & & \sigma_n^{N-t} \end{pmatrix}$$

- if $\sigma_i > 1$ exponential growth, exploding gradient
- if $\sigma_i < 1$ gradient vanishing

Exploding gradient

- Gradient values > 1 at every iteration
- Weak performances
- *NaN* everywhere quickly !

$\Rightarrow$ **Gradient clipping**

$$\nabla_W E = \begin{cases} \frac{\nabla_W E}{\|\nabla_W E\|} & \text{if: } \|\nabla_W E\| > 1 \\ \nabla_W E & \text{else} \end{cases}$$

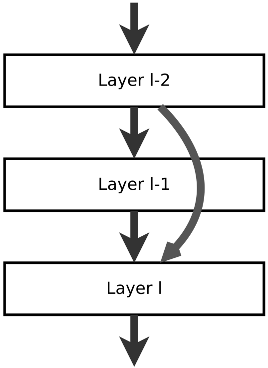
```
# use before 'optimizer.step()'
nn.utils.clip_grad_norm_(
    model.parameters(),
    max_norm=1.,
    norm_type=2
)
```

Vanishing gradient ⇒ Residual architecture (e.g. ResNet)

$$f(x) = x + f_0(x)$$
$$\frac{\partial f(x)}{\partial x_j} f(x) = 1 + \frac{\partial f_0(x)}{\partial x_j}$$

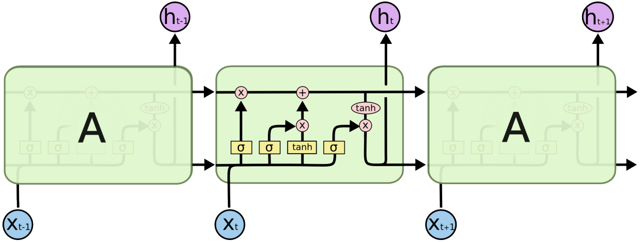
In the back propagation process:

$$\frac{\partial E_f(x)}{\partial x_j} f(x) = \frac{\partial E_y}{\partial y_j} + \sum_i \frac{\partial E_y}{\partial y_i} \frac{\partial y_i}{\partial x_j}$$



Vanishing gradient ⇒ gated unit (LSTM)

The phenomenon has been understood & (partially) overcome:
Neurons **learn** what should be **kept in memory** and what should be **forgotten**



Gated architecture

S. Hochreiter, J. Schmidhuber, Neural computation 1997
Long short-term memory

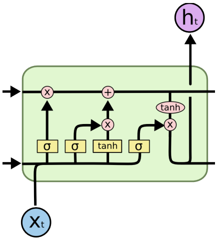
Chris Olah's blog `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Vanishing gradient ⇒ gated unit (LSTM)

The phenomenon has been understood & (partially) overcome:
Neurons **learn** what should be **kept in memory** and what should be **forgotten**

- Memory c_t (modified by simple operators)
- External representation h_t extracted from c_t
- Gates $y = \sigma(p) \otimes x$,

$\otimes$: term by term product & σ: sigmoid

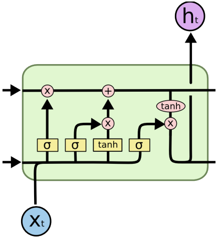


Vanishing gradient $\Rightarrow$ gated unit (LSTM)

The phenomenon has been understood & (partially) overcome:
Neurons **learn** what should be **kept in memory** and what should be **forgotten**

- Memory c_t (modified by simple operators)
- External representation h_t extracted from c_t
- Gates $y = \sigma(p) \otimes x$,

$\otimes$: term by term product & σ: sigmoid



Gate 1 (forgetting):

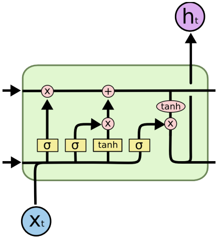
$$g_t^{(f)} = \sigma[W_f(x_t \oplus h_{t-1}) + b_f]$$
$$c_t^{(f)} = g_t^{(f)} \otimes c_{t-1}$$

Vanishing gradient ⇒ gated unit (LSTM)

The phenomenon has been understood & (partially) overcome:
Neurons **learn** what should be **kept in memory** and what should be **forgotten**

- Memory c_t (modified by simple operators)
- External representation h_t extracted from c_t
- Gates $y = \sigma(p) \otimes x$,

$\otimes$: term by term product & σ: sigmoid



Gate 2 (memory update):

$$g_t(i) = \sigma[W_{i,g}(x_t \oplus h_{t-1}) + b_{i,g}]$$

$$v_t(i) = \tanh[W_{i,v}(x_t \oplus h_{t-1}) + b_{i,v}]$$

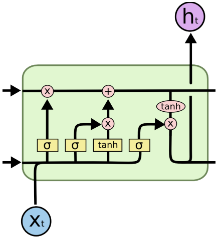
$$c_t = c_t^{(f)} + g_t^{(i)} \otimes v_t^{(i)}$$

Vanishing gradient $\Rightarrow$ gated unit (LSTM)

The phenomenon has been understood & (partially) overcome:
Neurons **learn** what should be **kept in memory** and what should be **forgotten**

- Memory c_t (modified by simple operators)
- External representation h_t extracted from c_t
- Gates $y = \sigma(p) \otimes x$,

$\otimes$: term by term product & σ: sigmoid



Gate 3 (output):

$$g_t(o) = \sigma[W_{o,g}(x_t \oplus h_{t-1}) + b_{o,g}]$$

$$v_t(o) = \sigma[W_{o,v}c_t + b_{o,v}]$$

$$h_t = g_t^{(o)} \otimes v_t^{(o)}$$

Conclusion

- **LSTM** and variation (**GRU**) are widely used
- biRNN (biLSTM / BiGRU) offer often faster convergence
- Gradient is always an issue: **watch it!**

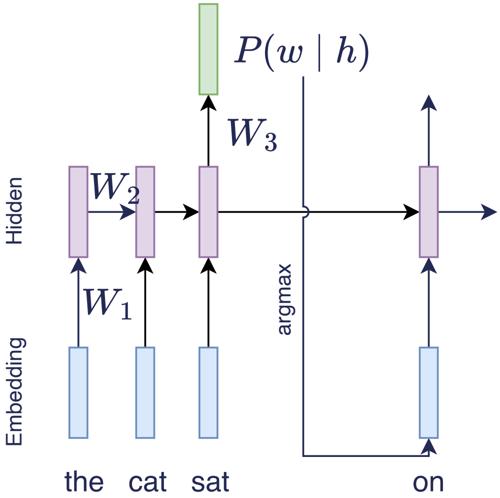
```python
# Calcul de || \nabla_W E ||^2
with torch.no_grad():
  grad_sql2 = sum((p.grad ** 2).sum() for p in l.parameters() )
```

- **Dropout** is useful to fight against overfitting
- Layers can be stacked (!) ⇒ `num_layers`

```python
# Module definition
model = torch.nn.LSTM(input_size=100, hidden_size=50,
                      batch_first=False, num_layers=1,
                      dropout=.3)
```

Data Generation

Basic data generation



Word prediction

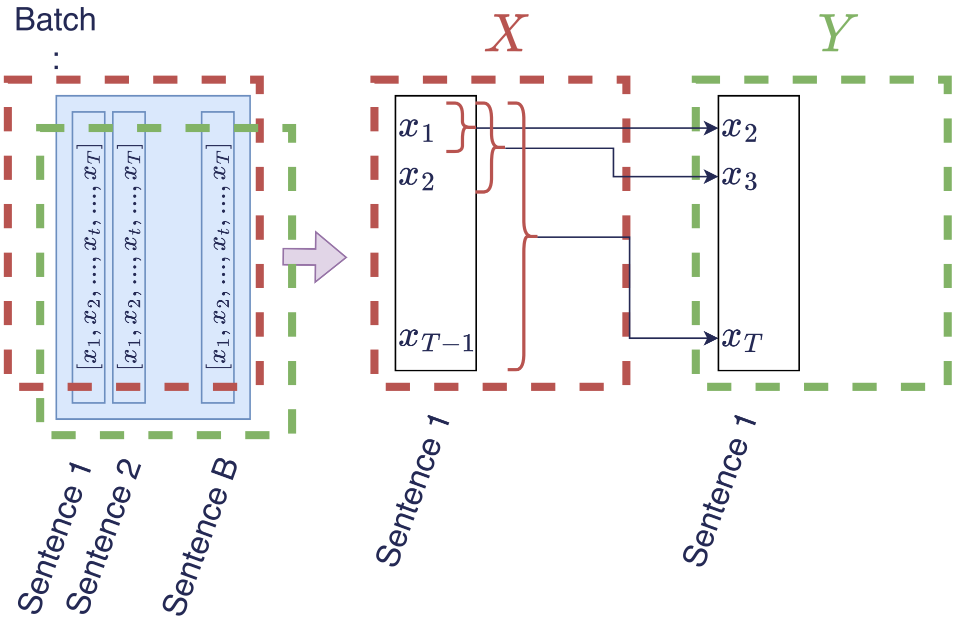
- $W_3 \approx$ vocabulary classification
- Several variation: argmax / sampling...

$\Rightarrow$ reduce vocabulary

  - at the letter scale (dim $\approx$ 60)
  - byte pair encoding $\Rightarrow$ most freq. ngrams of letters

- Iteration at $t + 1$ only requires:
  - x_{t+1}
  - h_t (that encode the whole sequence of previous words)

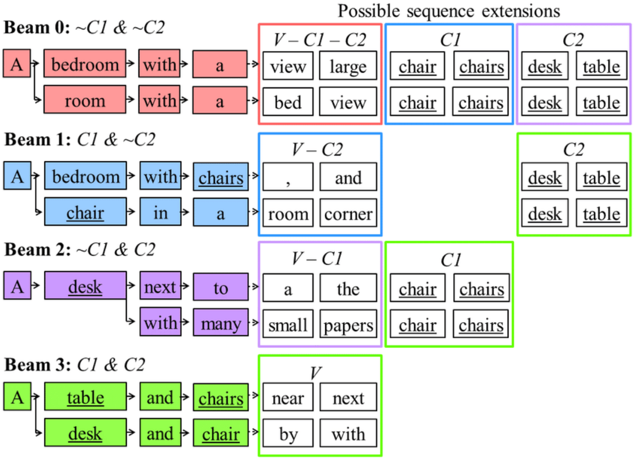
RNN In practice: teacher forcing & data shape



Beam search

Very difficult to find the best word at time t...

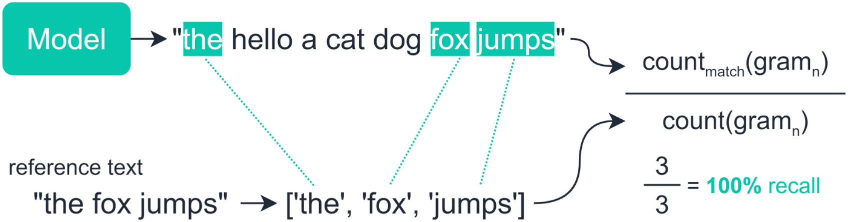
$\Rightarrow$ explore several possibilities, keep the most likely !



Evaluation: BLEU/ROUGE (translation & summary)

How to evaluate text generation?

Very difficult... still an open issue !

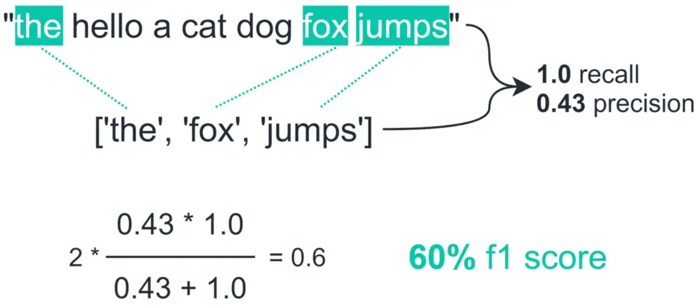


- f1 computations
- on N-grams, on longest common subsequence...

Evaluation: BLEU/ROUGE (translation & summary)

How to evaluate text generation?

Very difficult... still an open issue !



- f1 computations
- on N-grams, on longest common subsequence...

Conclusion

- Evaluation in the latent-space (cf BERT-score / CLS)
- Content evaluation (entity, figures...) $\Rightarrow$ hallucination problems