Réseaux récurrents et TAL

RCP 217

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers
Cédric, équipe Vertigo

2020–2021

Traitement des textes et séquences

- Classification : séquence $\rightarrow$ valeur ;
  - anti-spam ;
  - *sentiment analysis* ;
  - attribution d'auteur ;
- annotation : séquence (longueur n) $\rightarrow$ séquence (longueur n) ;
  - tagging / lemmatisation ;
  - reconnaissance d'entités nommées...
- réécriture : séquence (longueur n) $\rightarrow$ séquence (longueur **m**) ;
  - traduction automatique ;
  - résumé ;
  - analyse syntaxique

Un exemple de tâche : l'annotation

On a un texte, dont on cherche à annoter chaque « mot » :

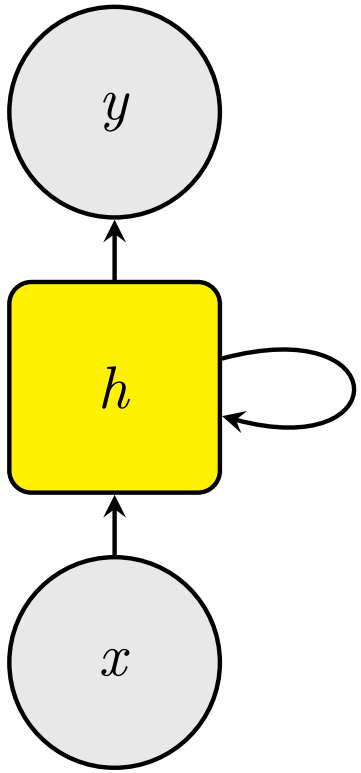
- parties du discours « lemmatisation » :

x	la	course	est	engagée
y	DET	NOUN	AUX	PART

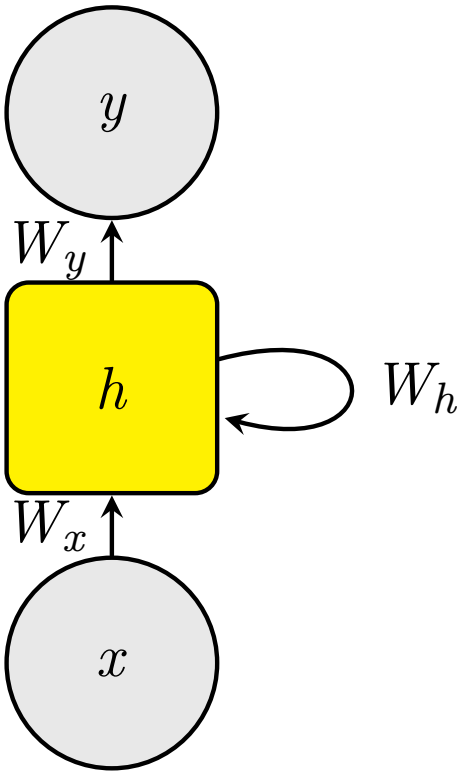
- En pratique :
  - en entrée, tenseur de forme $B \times N$ ($N =$ longueur de l'entrée) ;
  - en sortie, tenseur de forme $B \times N \times L$, où L est le nombre d'étiquettes possibles ;
  - autant de classificateurs que de sorties ; chacun est typiquement traité par un *softmax.*

Les réseaux récurrents

- Idée : on a une série d'entrées $x_1, x2...x_n$, et une série de sorties $z_1, z_2, ...z_n$;
- z_i dépend non seulement de x_i, mais aussi de $x_0, x_1, ..., x_{i-1}$;
- on utilise un réseau où la valeur de la couche cachée h est réinjectée comme entrée à la couche cachée de l'étape suivante.

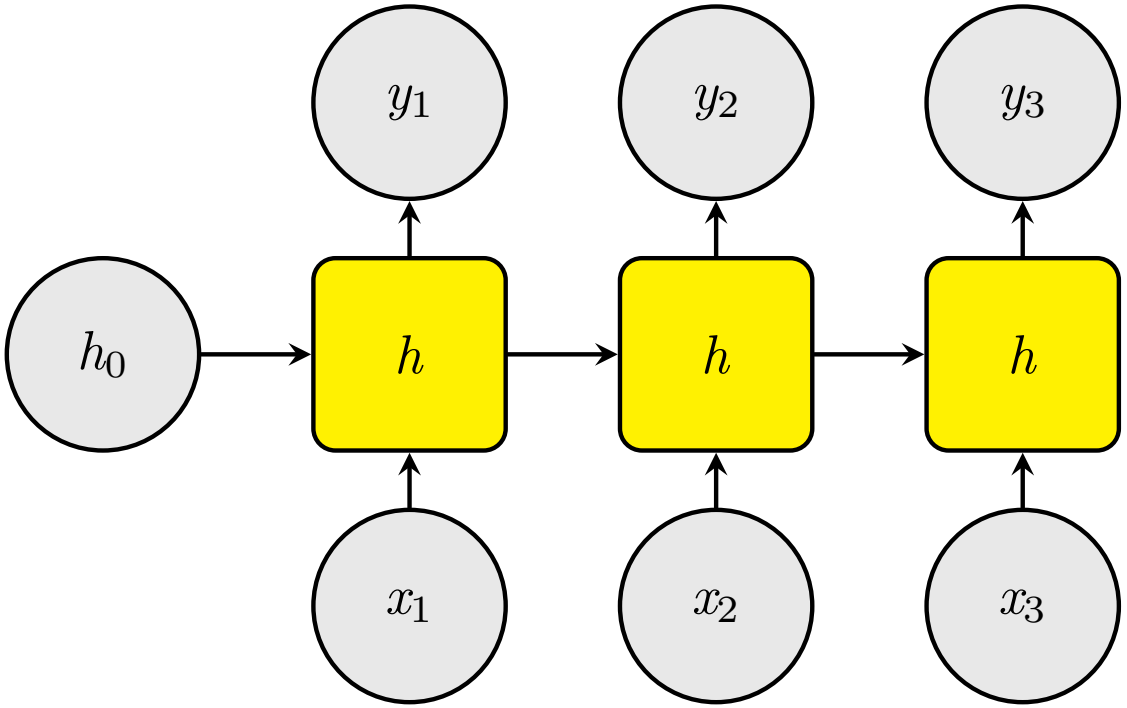


Les réseaux récurrents



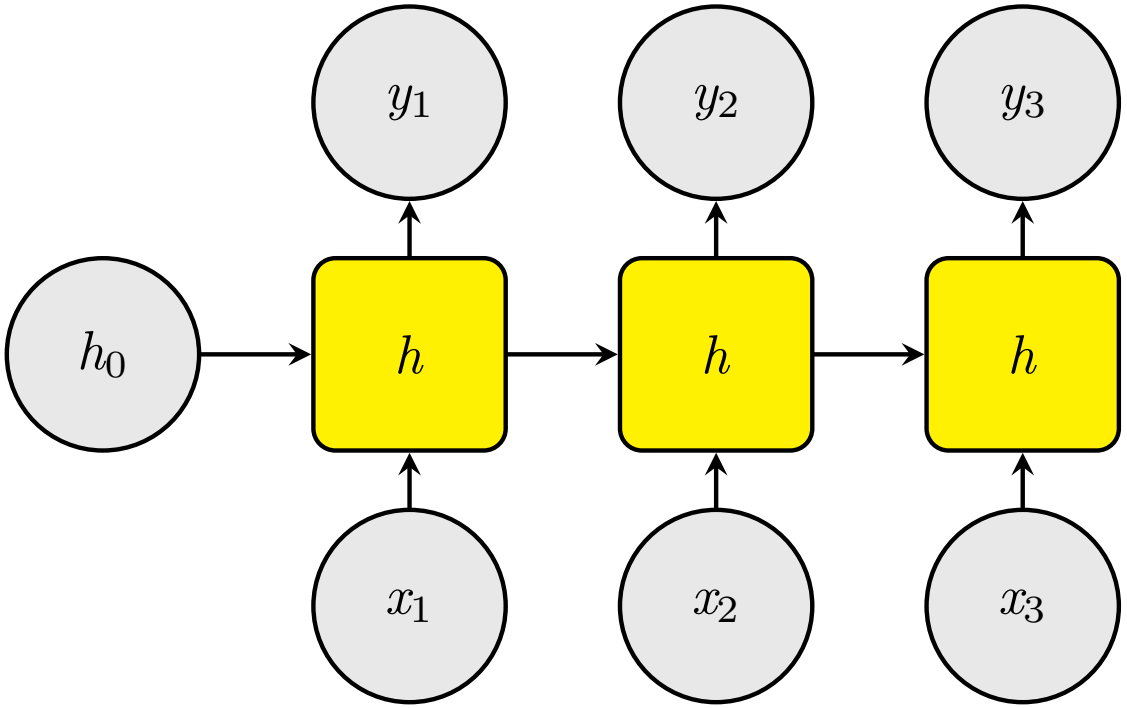
$$
\begin{aligned}
z_i &= W_x x_i + W_h h_{i-1} \\
h_i &= \sigma(z_i) \\
y_i &= W_y h_i
\end{aligned}
$$

Réseaux récurrents : vue dépliée



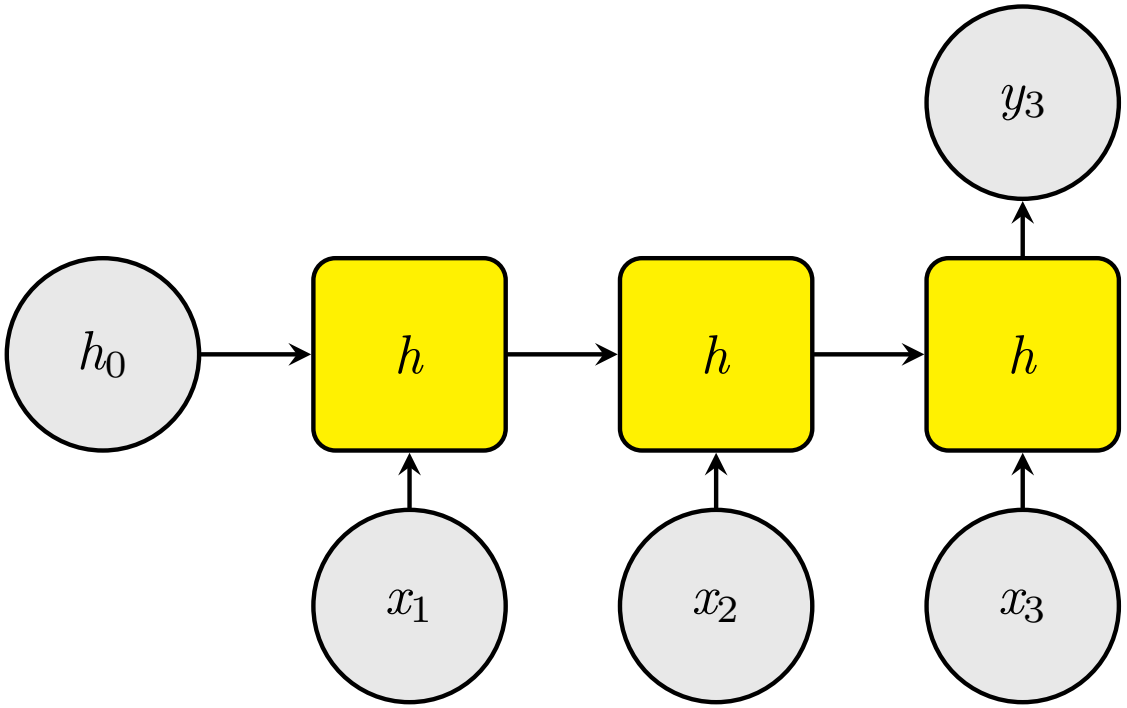
- on peut dessiner le réseau quand il travaille sur une entrée $(x_1, x_2, \ldots x_n)$
- la sortie de l'élément récurrent h nous donne potentiellement une sortie y_i pour chaque x_i ;

Réseaux récurrents : utilisation en annotation



- en annotation, la sortie du réseau pour l'entrée $(x_1, x_2, \ldots x_n)$ est $(y_1, y_2, \ldots y_n)$

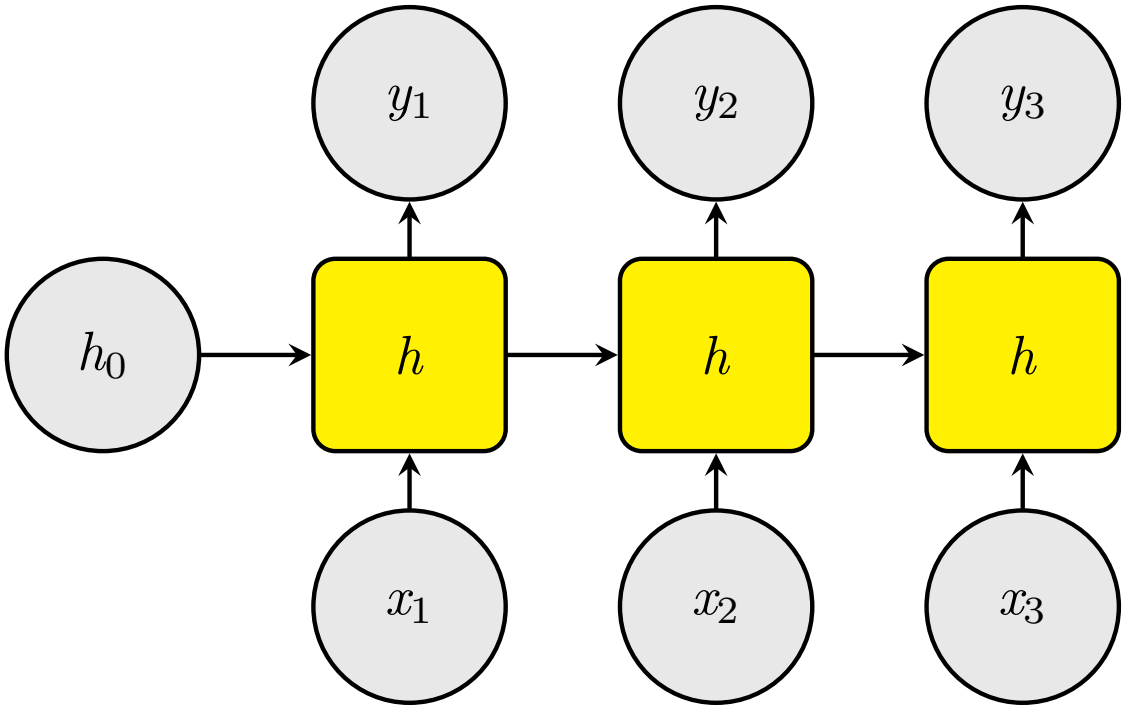
Réseaux récurrents : utilisation en classification



en classification, on ne prend que la *dernière* sortie : la sortie du réseau pour l'entrée $(x_1, x_2, \ldots x_n)$ est y_n

Réseaux récurrents : apprentissage

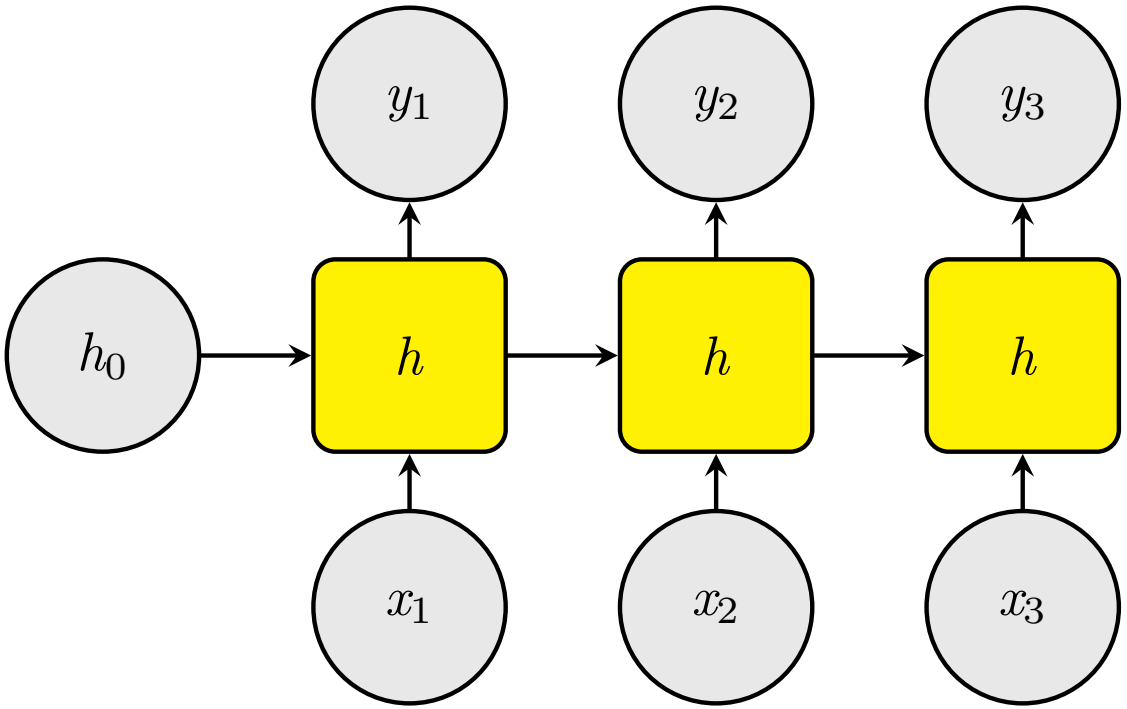
Sur une entrée :



- Fonction de coût pour une itération : $E(\hat{y}_i, y_i^*)$
- typiquement : $E(\hat{y}, y^*) = \sum_{i=1}^{n} E(\hat{y}_i, y_i^*)$
- pour la classification, simplement : $E(\hat{y}, y^*) = E(\hat{y_n}, y_n^*)$

Réseaux récurrents : apprentissage

Sur une entrée :



- ce réseau « déplié » est équivalent à un réseau multicouche, avec des poids partagés ;
- on peut calculer le gradient de l'erreur par rétropropagation ;
- et l'entraîner avec n'importe quelle méthode d'optimisation usuelle.
- plus n est grand, plus le réseau est profond.

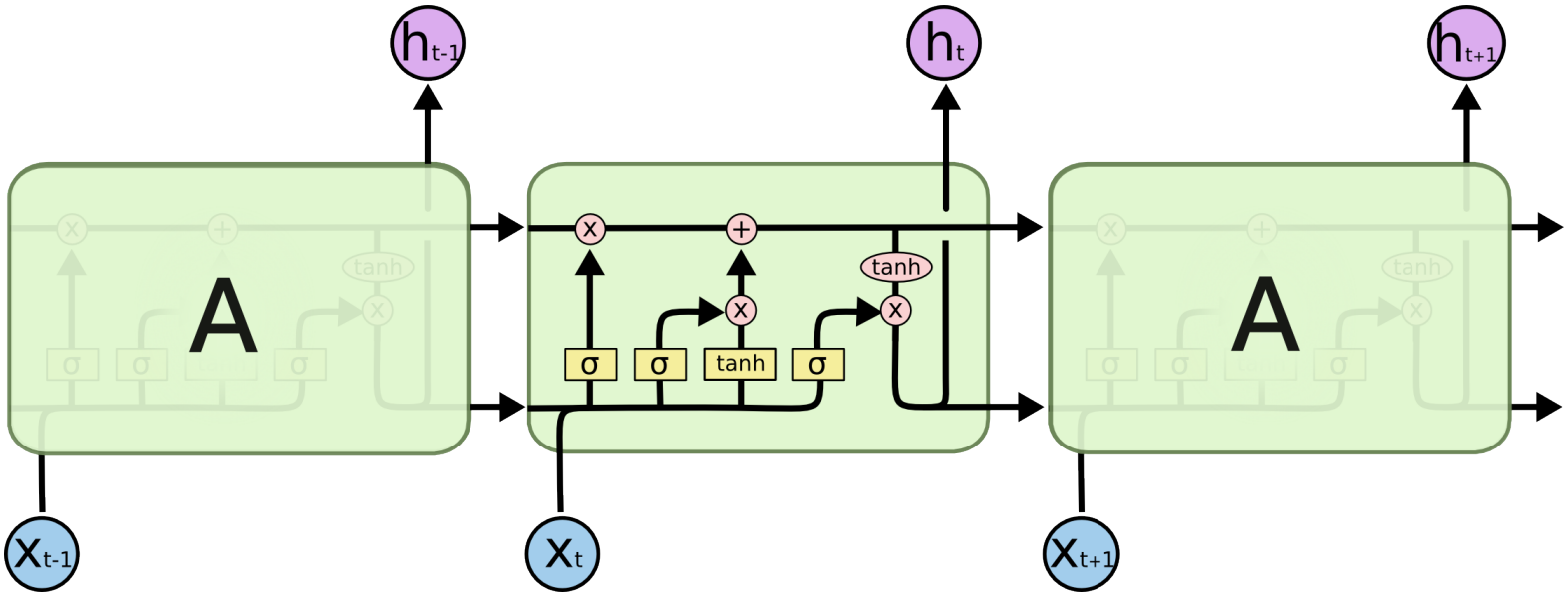
Vanishing/Exploding gradient

- Dans un réseau profond, le calcul des gradient implique la multiplication d'un grand nombre de facteurs ;
- sur des petits facteurs, on risque de trop s'approcher de 0 pour la précision de la machine (vanishing gradient) ;
- sur de grands facteurs, on risque d'obtenir des gradient trop importants, et de diverger (exploding gradient) ;
- un réseau récurrent a donc du mal à apprendre sur une séquence longue.

Long Short-term memory

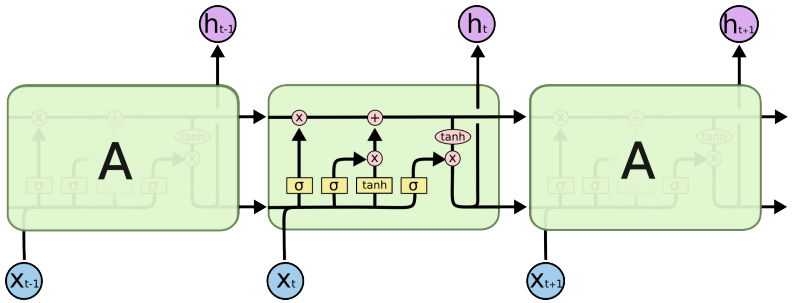
Idée

En plus du chemin « profond », ajouter un second chemin, reposant moins sur l'appel imbriqué de fonctions. Ce chemin sert de « mémoire » au système.



D'après Christopher Olah,
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-term memory



$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \mathsf{tanh}(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= ot \odot \tanh(c_t)
\end{aligned}
$$

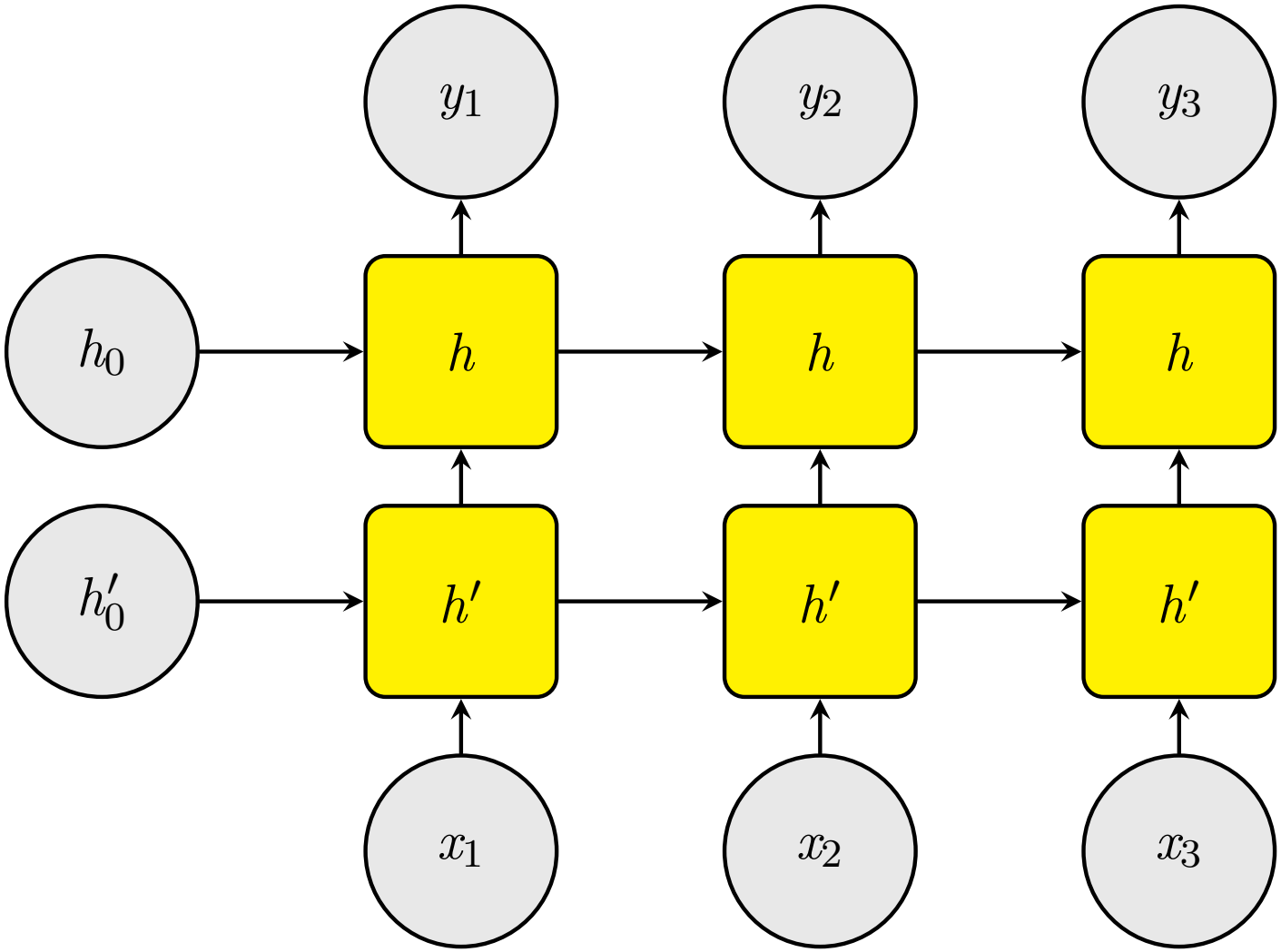
- c_t : mémoire du système ;
- f_t sert à *oublier sélectivement* de l'information ;
- i_t au contraire permet la mémorisation de nouvelles informations.

Gated Recurrent Units

- plus simples que les LSTM ;
- plus rapides pour les calculs et s'entraînent plus vite ;
- peut-être moins efficaces dans certains cas (the jury seems to be still out)

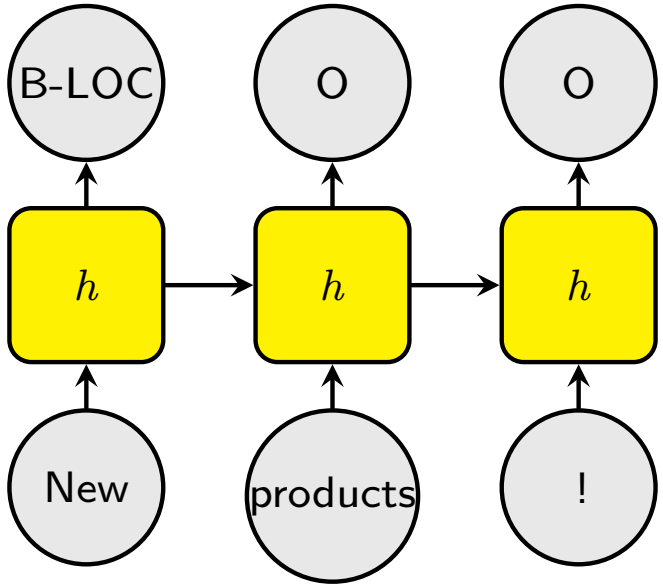
Réseaux multicouches

On peut empiler quelques couches de LSTM/GRU/RNN. Mêmes bénéfices que pour les réseaux classiques. En pratique : 2 ou 3 couches.



Réseau bidirectionnel

La tâche est la reconnaissance d'entités nommées.

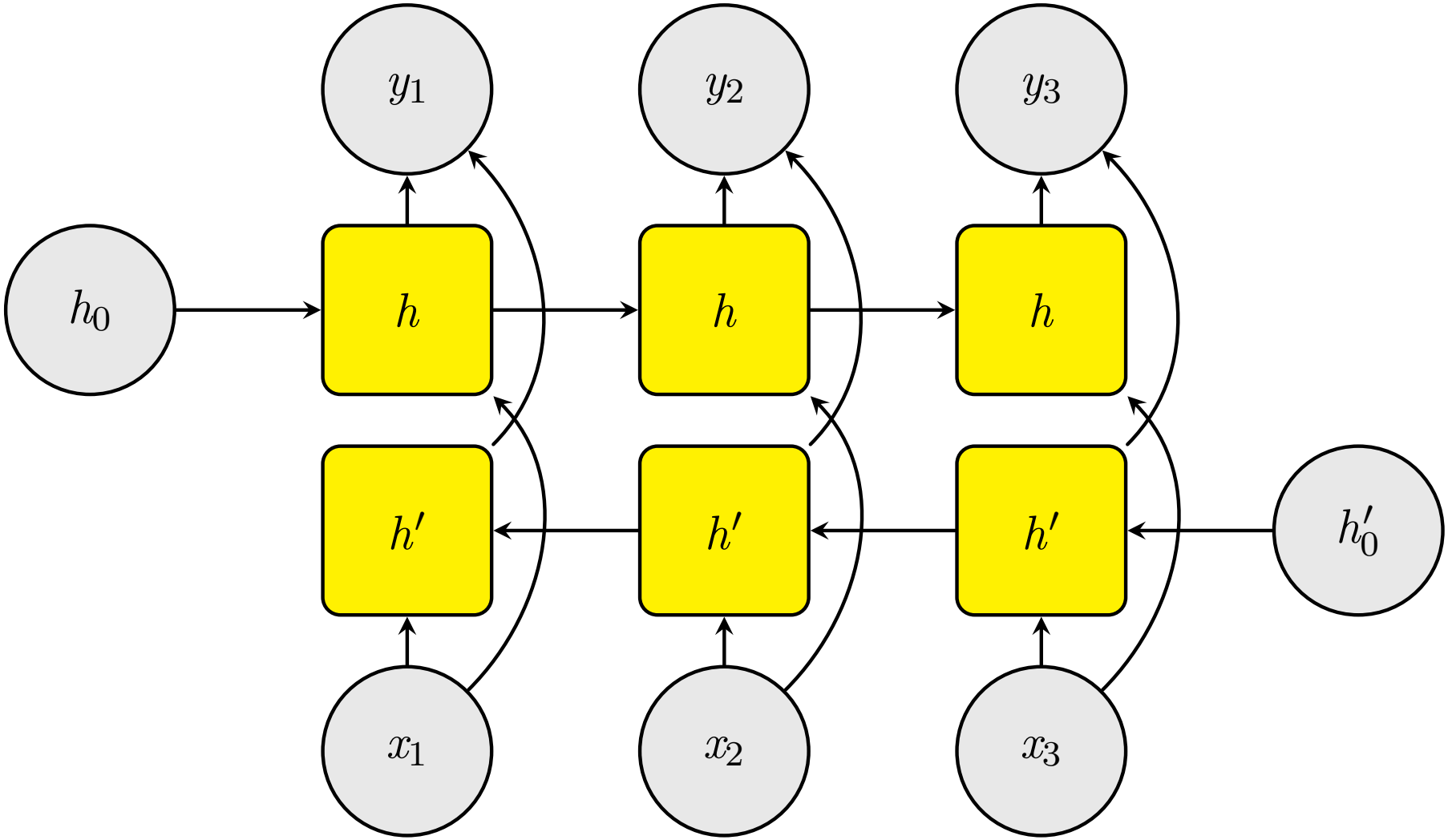


- la première cellule voit **uniquement** « New », et prédit une localité comme « New York » ;
- l'information de la seconde cellule permettrait de lever l'ambiguïté ;
- d'où les réseaux bi-directionnels.

Réseaux bi-directionnels

- Dans notre exemple précédent, nous avons dit que nous modélisions $P(Y_i = y_i | x_0, .., x_i)$;
- mais en fait, **on connaît tout** x
- on voudrait pouvoir modéliser au moins $P(Y_i = y_i | x_0, .., x_i, ...x_n)$;
- idée : utiliser un *second* LSTM, qui prend comme entrée $x_n, x_{n-1}, ...x_1$;
- pour chaque i, combiner les données de sortie des deux LSTM (par sommation ou concaténation).

Réseaux bi-directionnels



Les couches de pytorch (ex. pour les LSTM)

Deux variantes :

- **LSTM** : une couche optimisée pour le traitement des séquences :
  - entrée : $x, (h_0, c_0)$ où x est de la forme (B, N, M) (si `batch_first` est vrai) ;
  - sortie : $y, (h_n, c_n)$ où y est de la forme $(B, N, D * H)$
  - B : taille du batch, N : longueur de l'entrée, M dimension de l'entrée, H dimension de la couche cachée, D : nombre de directions (1 ou 2).
  - Quand on travaille étape par étape, par exemple pour un *modèle du langage*, il peut être intéressant de manipuler directement h_i et c_i. On peut aussi préférer utiliser **LSTMCell**.
- **LSTMCell** : une cellule de LSTM, on travaille sur *une seule étape* :
  - entrée : $x, (h, c)$ où x est de la forme (B, M) ;
  - sortie : $y, (h', c')$ où y est de la forme $(B, D * H)$;
  - pour traiter une séquence : boucle, en réinjectant les h' et c'.

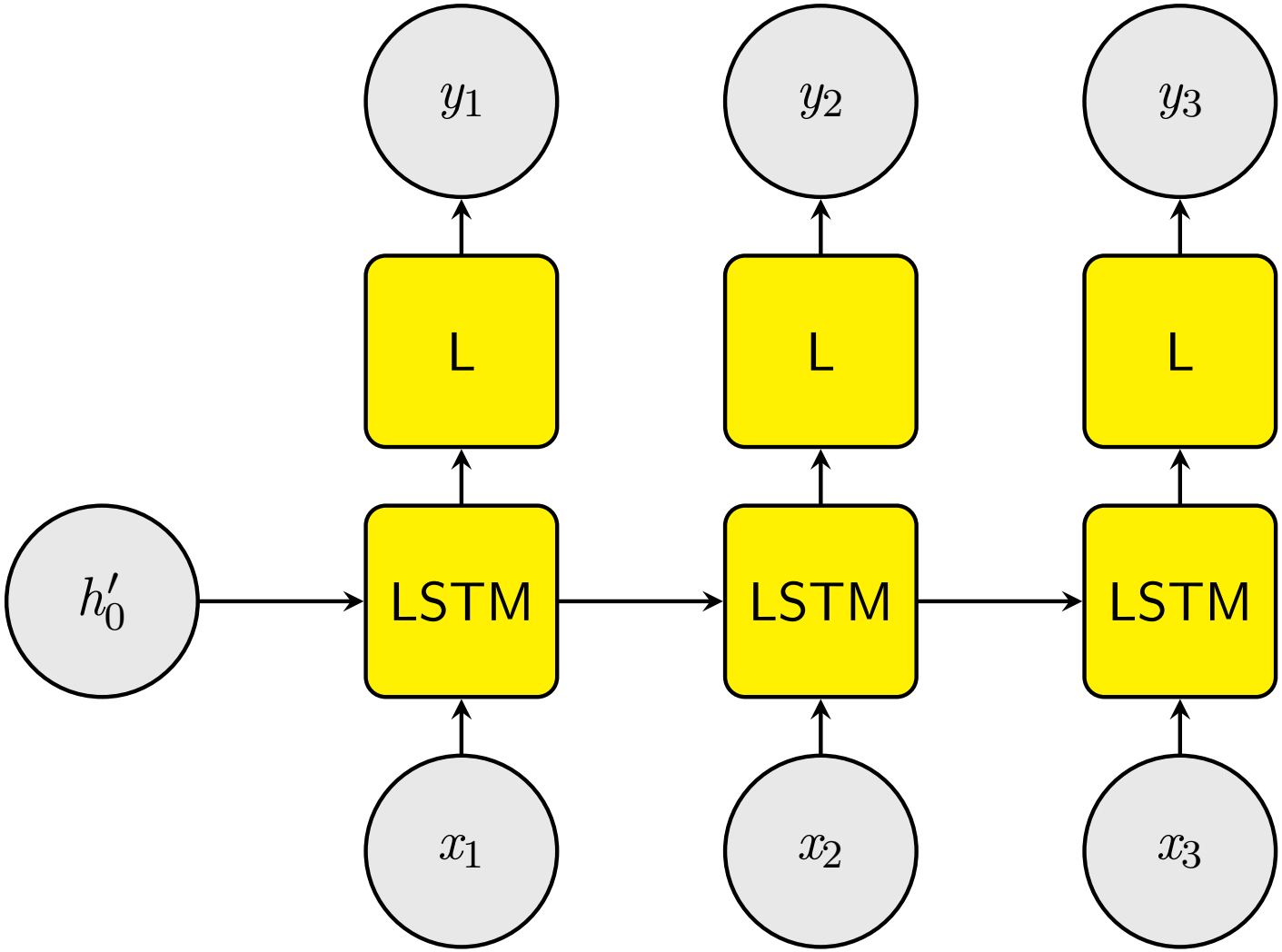
Exemple

```
self.embedding = Embedding.from_pretrained(vocabulaire.vectors,
                                           padding_idx=1,
                                           freeze=True)
self.recurrent = nn.LSTM(input_size=200, hidden_size=500,
                                            num_layers=2,
                                            batch_first=True, dropout=0.1)
self.linear_fin = nn.Linear(in_features=500, out_features=256)
```

Remarque

Une couche non récurrente combinée à un LSTM se combine par défaut *à chaque sortie*
Si on veut une seule sortie : utilisation de fonctions comme `torch.sum` ;

Couche non récurrente après un LSTM



Pourquoi le batch en second par défaut ?

Si on ne précise pas `batch_first`, le batch devient la seconde dimension, contrairement aux autres frameworks.

- pour des raisons de compatibilités ascendante ;
- parce que ça permet éventuellement d'itérer facilement sur N :

```
for t, out_t in enumerate(my_tensor) :
  pass
```

(d'après réponse sur stackoverflow)

Batch et padding

- généralement, les entrées sont de longueur différentes ;
- on utilise généralement du *padding* pour les lignes trop courtes ;
- ça fait probablement des calculs en trop, et ça peut être problématique pour la rétropropagation.
- Solution : les « packed sequences »

```
res = pack_padded_sequence(x, taille, batch_first=True,
                           enforce_sorted=True)
res, _ = self.lstm(res)
res, _ = pad_packed_sequence(res, batch_first=True)
```

- les « packed sequences » sont composées de deux parties : les données, et leur longueur ;
- elles sont utilisables par toutes les couches récurrentes de Pytorch.

Gradient Clipping

En cas *d'exploding gradient*, une solution simple est de le réduire « à la main » avant de procéder à l'optimisation des paramètres :

```
from torch.nn.utils import clip_grad_norm_
optimizer.zero_grad()
loss = model(x)
loss.backward()
clip_grad_norm_(model.parameters())
optimizer.step()
```

« Modèle du langage »

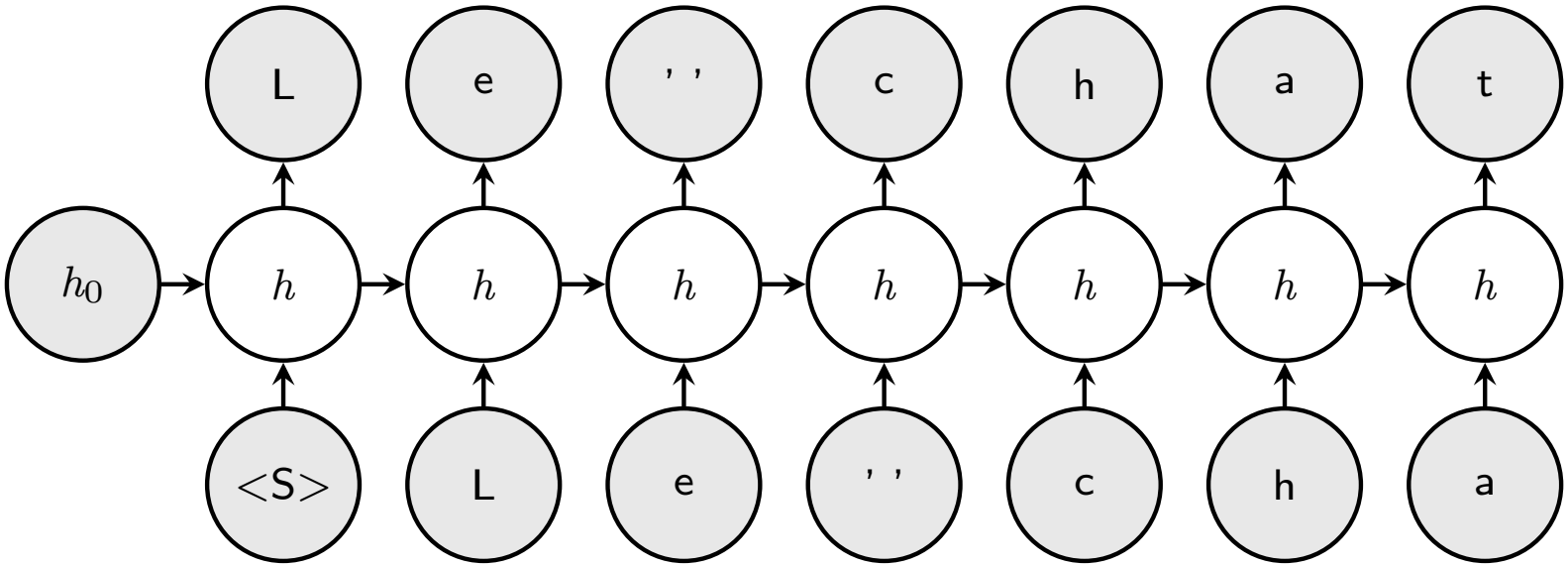
Définition

En Traitement Automatique des Langues, un *modèle du langage* est une fonction qui fournit une évaluation de $P(x)$, où x est une phrase dans la langue.

- but : évaluer $P(X_i = a | x_0, x_1 .., x_{i-1})$;
- utilité : saisie prédictive, choix de la traduction la plus « naturelle » en traduction automatique, de l'OCR le plus plausible...
- utile pour tout modèle de séquence.

Modèle du langage et réseaux récurrents

- entrée : séquence de mots ou de lettres ;
- sortie : mot ou lettre au rang $n + 1$;
- noter le symbole <S>, sentinelle pour le début du texte ;
- on a aussi un symbole <F>, qui marque la fin du texte.



Exemples

Blog d'Andrej Karpathy : The Unreasonable Effectiveness of Recurrent Neural Networks

Génération de « texte » :

- on commence avec le symbole de début « <S> » ;
- on obtient en sortie du RNN une distribution de probabilité pour le prochain token ;
- on tire au hasard (selon cette distribution) le prochain token ;
- on itère...
- fonctionne aussi avec de la musique.

Exemple de résultat (Karpathy, entraîné sur Shakespeare) :

> *PANDARUS :*
> *Alas, I think he shall be come approached and the day*
> *When little srain would be attain'd into being never fed,*
> *And who is but a chain and subjects of his death,*
> *I should not sleep.*

Autre exemple de tâche : la classification de textes

- Entrée : un texte ;
- Sortie : une étiquette (type de texte, bon/mauvais, auteur du texte...) ; typiquement codé par un softmax.

Première approche : « bag of words »

On décide la taille du vocabulaire total V de taille n_v (on réserve une entrée pour « mot inconnu »)
On représente le texte par un vecteur D de dimension n_v avec D_i : nombre d'occurrences du mot i.

- $\rightarrow$ utilisation de réseaux multi-couches standards ;
- ne prennent pas en compte l'ordre des mots ! ;
- pas toujours grave.

Codage par un gros vecteur de taille fixe

- peu réaliste pour des textes avec une architecture classique
- on peut envisager un réseau convolutif (convolutions 1D et non 2D) ;
- ça peut fonctionner, mais la « géométrie » d'une séquence n'est pas celle d'une image : les segments de la séquence ne sont pas forcément tous équivalents.

Utilisation d'un réseau récurrent

- On procède comme pour l'annotation, mais on ne s'intéresse qu'à l'étiquette obtenue après la lecture du *dernier* mot ;
- ça fonctionne ;
- ça n'apporte pas forcément beaucoup par rapport à un *Bag of Words* ou à un réseau convolutif : il faut que l'ordre des mots soit particulièrement important pour la tâche à accomplir ;
- si le token est au niveau du caractère et non du mot, c'est peut-être plus intéressant.

Réseaux récurrents pour l'annotation

- N'importe quel type de couche peut convenir : RNN, LSTM, GRU... ;
- architecture de base :
  - une couche d'embedding pour les mots ;
  - une couche récurrente ;
  - en sortie, une couche linéaire qui envoie sur l'ensemble des étiquettes ;
  - la sortie est interprétée avec un *softmax* ;
  - en Pytorch, `CrossEntropyLoss` le prend en charge : on calcule le softmax *en dehors* du réseau.

```
model = nn.Sequential(
          nn.Embedding(100000,300),
          nn.ReLU(),
          nn.LSTM(300,400)
          nn.Linear(400,8)
        )
```

Exploitation du résultat

Comment exploiter le résultat de l'annotation ?
Le `softmax` nous fournit une distribution de probabilités ;

- Algorithme glouton : on choisit à chaque fois l'annotation « la plus probable », indépendamment des précédentes.
- ça reste raisonnable : le softmax modélise $P(Y_i = y_i | x_0, .., x_i)$;
- mais on aurait à tout prendre préféré : $P(Y_i = y_i | x_0, .., x_i, y_0, ...y_{i-1})$;
- on peut alors utiliser une modélisation bi-gramme des séquences d'annotation (comme pour Markov Caché), et combiner les probabilités $P(Y_i = a | Y_{i-1} = b)$ avec les valeurs du obtenues par softmax - voire raisonner plutôt en terme de logarithmes, pour transformer les produits en sommes.
- meilleure solution : les CRF.

Exploitation du résultat (suite)

- si on combine un modèle interne (de la séquence à annoter) et un modèle de la séquence annotée elle même, on a un *Conditional Random Field*
- dans ce cas, on calcule le résultat par Viterbi ;
- attention, si on le fait brutalement, on a calculé la fonction de coût sans tenir compte des y_i sélectionnés - le coût calculé correspond à l'algorithme glouton ;
- voir le tutoriel de pytorch, ou, plus pratique, allenai/allennlp ou A PyTorch Tutorial to Sequence Labeling
- note : il existe aussi des algorithmes non neuronaux pour les CRF (logiciel Wapiti du Limsi)

Reconnaissance d'entités nommées

x	Victor	Hugo	partit	en	exil	à	Guernesey
y	B-NP	I-NP	O	O	O	O	B-GEO

Représentation **B-I-O**

B début d'entité nommée, ex. **B-NP** ;

I *suite* d'entité nommée, ex. **I-NP** ;

O élément hors d'une entité nommée ;

le système permet tout type d'annotation, pourvu que :

- un élément est connexe (un élément annote une suite de mot) ;
- les éléments sont disjoints (un mot appartient à un seul élément).

Limitations du modèle LSTM simple

Dans le modèle actuel :

- le choix de y_k ne dépend **que de** x ;
- mais on peut se retrouver avec une séquence y_{k-1}, y_k très peu probable, voire impossible ;
- problème assez mineur pour l'annotation des parties du discours ;
- mais majeur pour la reconnaissance d'entités nommées :

  O O I-PER

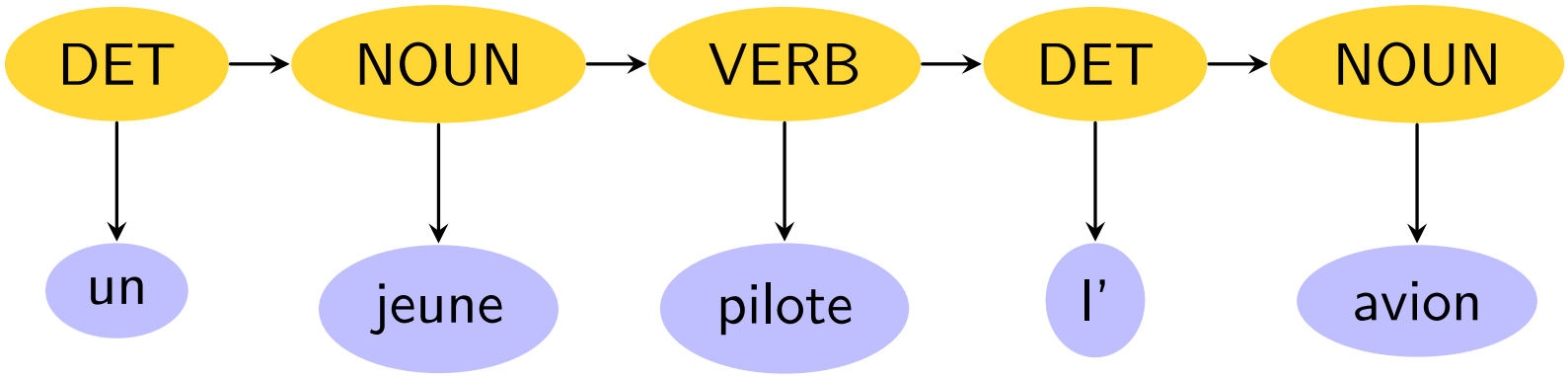
  est impossible ! un « I » suit toujours un « B ».
- on désire ajouter un système qui prend *aussi* en compte les séquences d'annotations ;
- ...et qu'on entraîne en fonction de cela.

Les Conditional Random Fields

Lafferty, McCallum, et Pereira, « Conditional Random Fields », 2001

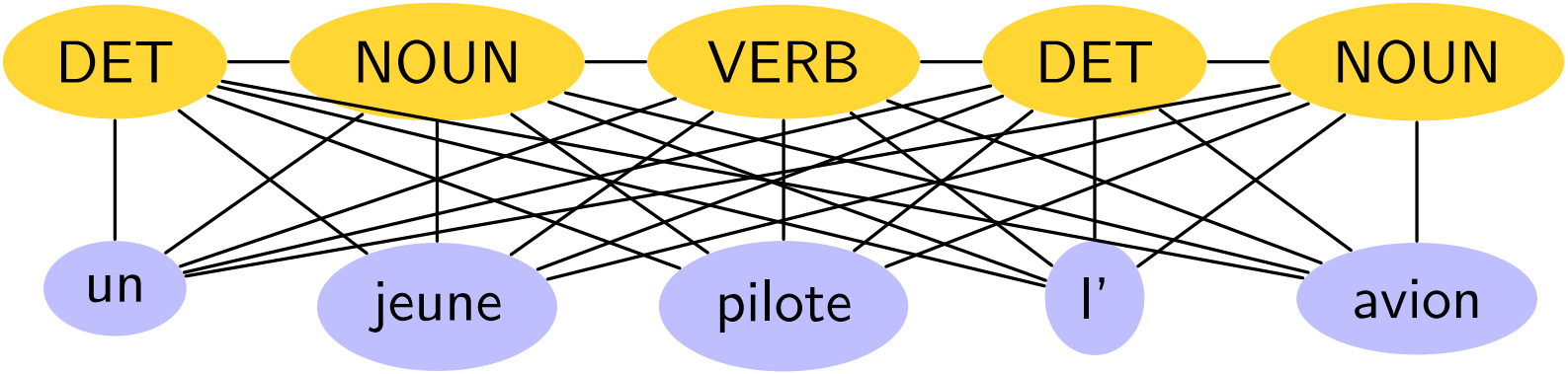
- on a une entrée $x = x_1 x_2 \cdots x_n$;
- on a une annotation $y = y_1 y_2 \cdots y_n$;
- on veut modéliser $p(y|x)$;
- dans un *conditional random field linéaire,* on fait intervenir uniquement des termes de la forme $f(y_i, y_{i-1}, x)$;
- $\rightarrow$ on modélise $p(y|x)$ en utilisant potentiellement **tout** x, mais, pour les y, on se contente **d'un voisinage limité ;**
- dans un *conditional random field* général, on construit un *graphe* entre les x_k et surtout les y_k possibles, et les fonctions utilisées portent uniquement sur des nœuds voisins dans le graphe ;
- dans le cas linéaire, le graphe relie : y_k et y_{k-1}, et chaque y_k à *tous* les x_i.

Markov caché



- modèle très sommaire ;
- prend en compte le label précédent et la relation label/mot ;
- viterbi permet un calcul *global* de la probabilité ;
- gère en partie les phrases *garden-path* ;
- mais pas d'information lexicale entre les mots : n'utilise pas l'affinité sémantique entre *pilote* et *avion*.

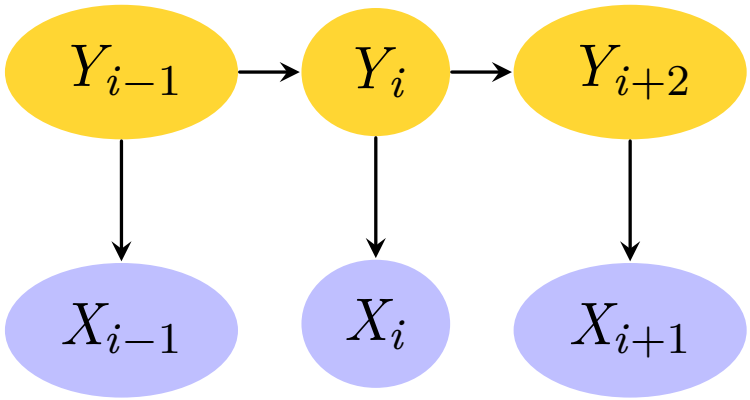
Conditional Random Fields



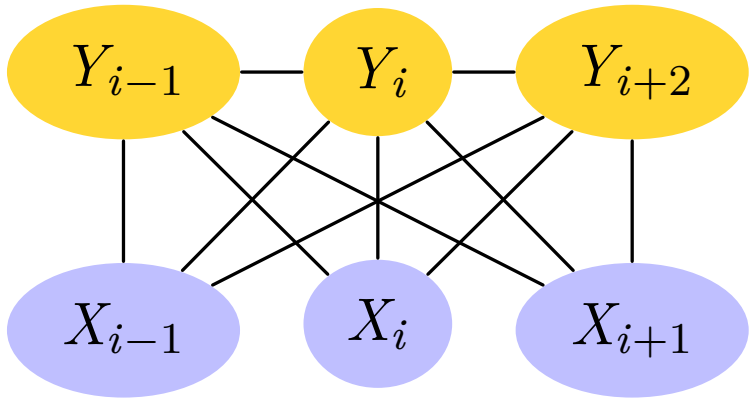
- les Y peuvent dépendre de n'importe quel X ;
- le graphe peut éventuellement être plus complexe (mais au prix de calculs plus coûteux) ;
- des systèmes comme wapiti peuvent travailler sur des *propriétés* arbitraires des mots (capitalisation, terminaison, etc...) ;

Markov caché et Conditional Random Fields

Réseau de Markov Caché



Conditional Random Field



- les Y ne « créent » plus les X (pas un modèle « génératif » mais « discriminant » ;

Modélisation des CRF

- si $\mathcal{X}$ est l'ensemble des mots, et $\mathcal{Y}$ est l'ensemble des annotations,
- si $x = x_1 x_2 \cdots x_n \in \mathcal{X}^n$, $y = y_1 y_2 \cdots y_n \in \mathcal{Y}^n$ et $y_0 = <\text{START}>$
- On pose :

$$p(y|x) = \frac{\exp\left(\sum_{i=1}^{n} f(y_i, y_{i-1}, x)\right)}{Z(x)}$$

- où $Z(x) = \sum_{y' \in \mathcal{Y}^n} \exp\left(\sum_{i=1}^{n} f(y'_i, y'_{i-1}, x)\right)$ est la **fonction de partition ;**
- $Z(X)$ garantit que p est effectivement une distribution de probabilités.

CRF et réseaux récurrents

- on appelle $h(x_i)$ la sortie du réseau récurrent qui correspond à l'entrée x_i ;
- $h(x_i)$ est un vecteur de dimension $|\mathcal{Y}|$;
- $h_t(x_i)$ est la composante t du vecteur, qui concerne $t \in \mathcal{Y}$;
- on définit par ailleurs une matrice A dans $\mathbb{R}^{|\mathcal{Y}|\times|\mathcal{Y}|}$ qui décrit les transitions entre deux labels t_1 et t_2 : $A_{t_1 t_2}$ est d'autant plus grand que la séquence de labels $t_1 t_2$ est plausible.
- on pose alors :

$$p(y|x) = \frac{\exp\left(\sum_{i=1}^{n} A_{y_{i-1},y_i} + h_{y_i}(x_i)\right)}{Z(x)}$$

- l'apprentissage revient alors à **maximiser** $p(y|x)$ pour les observations.

Calcul de la fonction de partition

- on doit l'utiliser : sinon le calcul est erroné, et on risque de maximiser le numérateur pour *toutes* les valeurs, y compris les fausses !
- problème : dans $Z(x) = \sum_{y' \in \mathcal{Y}^n} \exp\left(\sum_{i=1}^{n} f(y'_i, y'_{i-1}, x)\right)$ on a $y' \in \mathcal{Y}^n$. Calcul « naïf » en $\mathcal{O}(|\mathcal{Y}|^n)$;
- mais calcul par programmation dynamique possible :

$$
\begin{aligned}
Z(x) &= \sum_{y' \in \mathcal{Y}^n} \exp\left(\sum_{i=1}^{n} f(y'_i, y'_{i-1}, x)\right) \\
&= \sum_{y' \in \mathcal{Y}^n} \prod_{i=1}^{n} \exp\left(f(y'_i, y'_{i-1}, x)\right)
\end{aligned}
$$

Rappel : pour que ça soit défini pour $i = 1$, on introduit un y_0 qui vaut toujours <START>.

Pour alléger l'écriture, posons $g(y'_i, y'_{i-1}) = \exp(f(y'_i, y'_{i-1}, x))$ et $Z = \sum_{y' \in \mathcal{Y}^n} \prod_{i=1}^{n} g(y'_i, y'_{i-1})$. Définissons la fonction $\alpha(k, t)$:

$$\alpha(k, t) \triangleq \sum_{y' \in \mathcal{Y}^k} \prod_{i=1}^{k} g(y'_i, y'_{i-1}) g(t, y'_k)$$

$$\alpha(0, t) = g(t, \text{<START>})$$

$$\alpha(k, t) = \sum_{y'_k \in \mathcal{Y}} \sum_{y' \in \mathcal{Y}^{k-1}} \prod_{i=1}^{k} g(y'_i, y'_{i-1}) g(t, y'_k)$$

$$= \sum_{y'_k \in \mathcal{Y}} \sum_{y' \in \mathcal{Y}^{k-1}} \prod_{i=1}^{k-1} g(y'_i, y'_{i-1}) g(y'_{k-1}, y'_k) g(t, y'_k)$$

$$= \sum_{y'_k \in \mathcal{Y}} g(t, y'_k) \alpha(k-1, y'_k)$$

récapitulation sur les CRF

On obtient :

- Initialisation : $\alpha(0, t) = g(t, <\text{START}>)$
- récurrence : $\alpha(k, t) = \sum_{y'_k \in \mathcal{Y}} g(t, y'_k)\alpha(k-1, y'_k)$
- $Z = \sum_{y'_n \in \mathcal{Y}} \alpha(n, y'_n)$
- Z se calcule donc en $\mathcal{O}(|\mathcal{Y}| \times n)$

En pratique, on travaille sur les $\log$ des $\alpha(k, t)$.

Représentation des mots par des LSTM

Idée : coupler en entrée

- un embedding classique (avec codage *one-hot* du mot) ;
- une représentation obtenue en faisant passer les lettres du mot par un LSTM ;

Avantage : meilleure représentation des mots inconnus ; alternative à Fasttext.

Bibliographie

- Hochreiter, et Schmidhuber. « Long Short-Term Memory ». 1997
- Lample, Ballesteros, Subramanian, Kawakami, et Chris Dyer. « Neural Architectures for Named Entity Recognition », 2016. https ://doi.org/10.18653/v1/N16-1030.
- Sutton, Charles. « An Introduction to Conditional Random Fields ». 2012.
- Yamada, Asai, Shindo, Takeda, et Matsumoto. « LUKE : Deep Contextualized Entity Representations with Entity-aware Self-attention ». 2020

Webographie

- allenai/allennlp Python. 2017. Reprint, AI2, 2021. *bibliothèque très riche pour le TAL*
- Arbel, Nir. « How LSTM Networks Solve the Problem of Vanishing Gradients ». Medium, 16 mai 2020.
- Karpathy, Andrej. « The Unreasonable Effectiveness of Recurrent Neural Networks ». Andrej Karpathy blog, 2015.
- Vinodababu, Sagar. A PyTorch Tutorial to Sequence Labeling 2021.