AMAL - TP 4
Réseaux récurrents

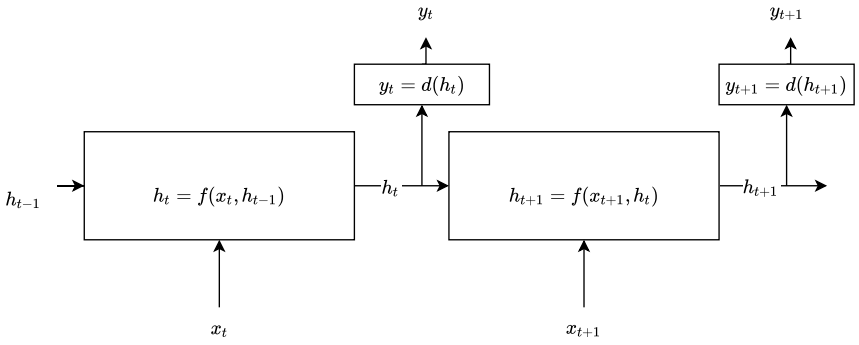
Nicolas Baskiotis - Benjamin Piwowarski - Laure Soulier

2020-2021

1 Introduction (brève, cf cours)

Les réseaux récurrents (RNN) sont un type d'architectures privilégié pour traiter les séquences. De façon générique, un réseau récurrent utilise un état caché (ou *mémoire*) pour accumuler l'information des itérations précédentes. Lors du calcul à un moment t de la séquence, le réseau utilise à la fois l'entrée à l'instant t et l'état caché pour inférer un nouvel état caché. L'état caché peut être décodé pour prédire une valeur de sortie correspondant à l'instant t mais également être utilisé pour continuer l'inférence au temps suivant.

Formellement, un réseau récurrent f calcule l'état suivant par une étape d'encodage : $\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$, avec $\mathbf{x}_t$ l'entrée à l'instant t, $\mathbf{h}_{t-1}$ l'état caché de l'instant précédent, $\mathbf{h}_t$ l'état caché produit par le réseau à partir de $\mathbf{x}_t$ et $\mathbf{h}_{t-1}$. Un décodeur d permet d'inférer la sortie d'intérêt $\mathbf{y}_t$ à partir de l'état caché $\mathbf{h}_t$.



Un exemple simple de réseau est $\mathbf{h}_{t+1} = f(\mathbf{x}_t, \mathbf{h}_t) = tanh(\mathbf{W}_i\mathbf{x}_t + \mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{b}_h)$ qui opère une transformation linéaire sur les entrées (*input*) définie par les poids $\mathbf{W}_i$, une sur l'état caché (*hidden*) définie par les poids $\mathbf{W}_h$, combine les deux puis applique une non-linéarité (tangente hyperbolique, sigmoïde ou ReLU par exemple). L'état caché peut être décodé également avec un réseau linéaire combiné à une non-linéarité : $\mathbf{y}_t = d(h_t) = \sigma(\mathbf{W}_d\mathbf{h}_t + \mathbf{b}_d)$.

Par exemple, soit $\mathbf{x} = \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$ une séquence en entrée (possiblement multivariée : $\mathbf{x}_i \in \mathbb{R}^d$) et $\mathbf{h}_0$ un état caché initial. Pour calculer le décodage de cette séquence, les étapes

sont les suivantes :

$$\begin{array}{llll} \mathbf{h}_1 = f(\mathbf{x}_1, \mathbf{h}_0) & \mathbf{h}_2 = f(\mathbf{x}_2, \mathbf{h}_1) & \dots & \mathbf{h}_T = f(\mathbf{x}_T, \mathbf{h}_{T-1}) \\ \mathbf{y}_1 = d(\mathbf{h}_1) & \mathbf{y}_2 = d(\mathbf{h}_2) & \dots & \mathbf{y}_T = d(\mathbf{h}_T) \end{array}$$

D'autres multiples possibilités existent pour calculer l'état caché : par exemple, avoir un réseau commun à la place de deux réseaux linéaires séparés pour les entrées et l'état. Dans ce cas, une concaténation des entrées et de l'état caché est opérée et une matrice de poids unique est utilisée sur ce vecteur concaténé. Nous verrons dans les TPs suivants des architectures plus complexes.

À partir de ce principe général, les RNNs sont très flexibles et de multiples applications existent :

— l'état caché peut être décodé uniquement à la fin d'une séquence pour produire une sortie unique. Ce type de réseau, appelé *many-to-one* (figure 1), permet par exemple de classifier une séquence : le décodage de la sortie indique alors la classe de la séquence ;

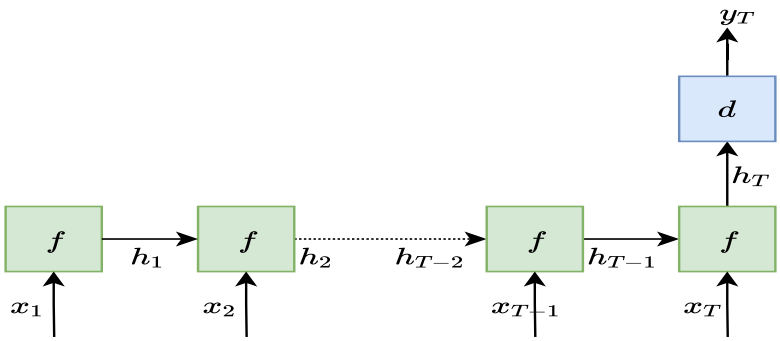


FIGURE 1 – Exemple de réseau many-to-one

— l'état caché peut être décodé à chaque pas de temps (type *many-to-many*, figure 2), pour du forecasting (prédiction de températures, d'embouteillages) ou de la génération de texte ;

— une variante (figure 3 consiste à lire toute une séquence puis produire une séquence à partir de l'état caché final (en particulier pour la traduction) ; il faut dans ce cas introduire lors de la génération de la nouvelle séquence l'élément décodé au pas d'avant $\mathbf{y}_t$ (comme il n'y a pas d'entrée $\mathbf{x}_t$ correspondante).

— il peut y avoir également qu'une seule entrée à partir de laquelle on produit une séquence (type *one-to-many*, figure 4).

Dans la suite de ce TP, nous étudierons successivement le problème de la classification de séquences (architecture de la figure 4), du forecasting (architecture 2) et enfin de la génération (architecture 2 également). Les autres architectures seront étudiées dans les TPs suivants.

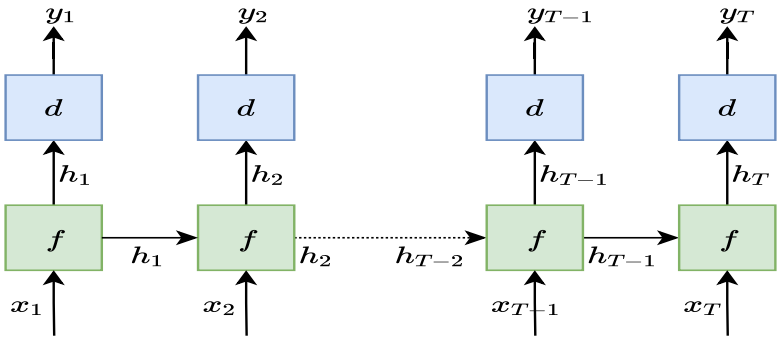


FIGURE 2 – Exemple de réseau many-to-many

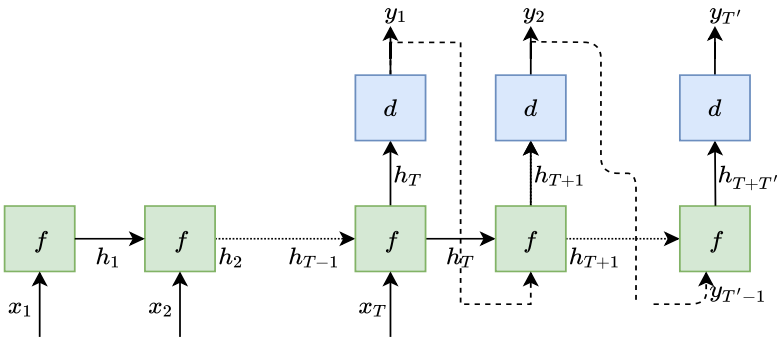


FIGURE 3 – Exemple de réseau many-to-many, variante

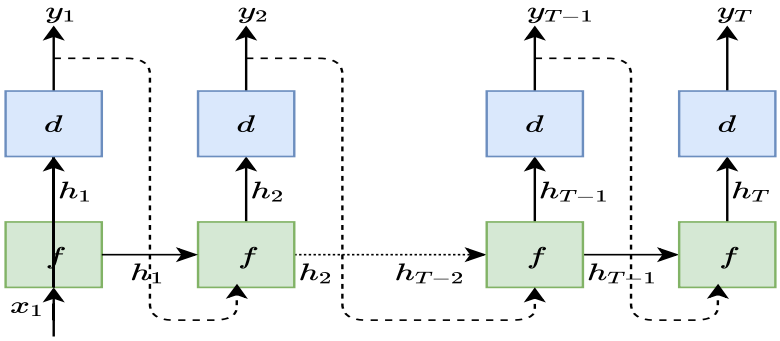


FIGURE 4 – Exemple de réseau one-to-many

Important

Dans toute la suite, nous considérerons que nos modèles peuvent prendre en entrée un batch de séquences. La dimension de l'entrée sera toujours $length \times batch \times dim$ avec $length$ la longueur des séquences, $batch$ la taille du batch et dim la dimension d'un élément de la séquence. Nous nous limiterons à des séquences de tailles fixes par batch (elles peuvent être de taille différentes pour différents batchs) afin de faciliter la constitution des batchs.

2 Implémentation d'un RNN

Question 1

Coder une classe `RNN` qui implémente un réseau récurrent simple tel que décrit en introduction.

Important

Les architectures présentées ci-dessus utilisent toutes le même modèle de RNN. La différence se fait au niveau de la supervision, de l'utilisation des états cachés et du décodage. Dans la suite, il vous est donc demandé d'implémenter **une seule classe RNN** et selon l'application visée, votre boucle d'apprentissage changera.

Pour un réseau dont l'état caché est de dimension `latent`, l'entrée de dimension `dim` et la sortie désirée de dimension `output`, la classe doit comporter :

- une fonction `one_step(x,h)` qui permet de traiter un pas de temps : elle prend en entrée un batch $\mathbf{x}$ à un instant t des séquences de taille $batch \times dim$ et le batch des états cachés de taille $batch \times latent$, et renvoie les états cachés suivants de taille $batch \times latent$.
- la méthode `forward(x,h)` qui doit traiter tout le batch de séquences passé en paramètre. Elle appelle successivement la méthode `one_step` sur toute les éléments des séquences. La taille de $\mathbf{x}$ est $length \times batch \times dim$ et $\mathbf{h}$ de taille $batch \times latent$. Elle doit renvoyer la séquence des états cachés calculés de taille $length \times batch \times latent$.
- une méthode `decode(h)` qui permet de décoder le batch d'états cachés, avec $\mathbf{h}$ de taille $batch \times latent$ et renvoie un tenseur de taille $batch \times output$.

3 Classification de séquences

Télécharger le jeu de données des températures. Chaque ligne représente la température chaque heure pour 30 villes américaines.

L'objectif est de construire un modèle qui à partir d'une séquence de température de longueur non prédéfinie infère la ville correspondante. Typiquement, ce genre de problème est impossible à traiter avec un réseau classique : même en considérant des séquences de longueur fixée, les dimensions ne sont pas homogènes en fonction des séquences (une dimension ne représente pas toujours la même heure).

Question 2

Tester votre implémentation sur le jeu de données, en sélectionnant un sous-ensemble d'une dizaine de villes.
Pour le décodage, comme l'objectif est de faire de la classification multi-classe, il est usuel d'utiliser une couche linéaire, suivie d'un softmax couplé à un coût de cross-entropie.
Vous considérerez pour chaque batch des séquences de même longueur. Attention toutefois à tirer aléatoirement le début des séquences d'apprentissage (afin de ne pas avoir un jeu de données biaisé qui commence toujours aux mêmes heures). Vous pouvez en bonus varier la taille des séquences selon les batchs. Par ailleurs, vous pouvez utiliser un tenseur nul pour l'état initial. Observer le coût en apprentissage et en test, ainsi que le taux d'erreurs (coût 0-1). Faites varier pour le test la longueur des séquences.

Dans ce contexte de réseau *many-to-one*, la supervision ne se fait qu'en bout de séquence (et non pas à chaque instant de la séquence), lorsque la classe est décodée. L'erreur associée est rétro-propagée sur tous les états traversés.

4 Modèle de séquences multi-variées

L'objectif de cette partie est de faire de la prédiction de séries temporelles : à partir d'une séquence de température de longueur t pour l'ensemble des villes du jeu de données, on veut prédire la température à $t+1$, $t+2$, ...

Question 3

Que doit-on changer au modèle précédent ? Quel coût est dans ce cas plus adapté que la cross-entropie ?
Pour quantifier à quel point la corrélation entre les séries temporelles est prise en compte, vous entraînerez deux types de modèle en considérant n villes :

- un RNN par série de température propre à une ville, soit n modèles différents qui prennent chacun une série dans $\mathbb{R}$ et produisent un décodage dans $\mathbb{R}$
- un RNN commun à toutes les villes qui prend une série dans $\mathbb{R}^n$ et prédit une série dans $\mathbb{R}^n$.

Faire les expériences en faisant varier l'horizon de prédiction (à $t+2$, etc.) et la longueur des séquences en entrée.

Dans ce contexte de réseau *many-to-many*, la supervision peut se faire à chaque étape de la séquence sans attendre la fin de la séquence. La rétro-propagation n'est faîte qu'une fois que toute la séquence a été vue, mais à un instant t, le gradient prend en compte l'erreur à ce moment (en fonction de la supervision du décodage) mais également l'erreur des pas de

temps d'après qui est cumulée.

5 Génération de séquences

Principe La génération de séquences consiste à produire une séquence de symboles discrets à partir d'une séquence en entrée. L'objectif est donc de décoder à partir de l'état caché une distribution de probabilités multinomiale sur les symboles à engendrer. Il faut donc que la dimension de sortie du RNN soit égale au nombre de symboles considérés. Il faut par ailleurs utiliser un softmax pour obtenir une distribution à partir du décodage de l'état caché. Le coût cross-entropie est adapté pour apprendre cette distribution.

Embedding Lorsque la séquence d'entrée est également discrète, il faut tout d'abord projeter cet espace d'entrée dans un espace continu - ce qu'on appelle un *embedding*. Supposons que l'on ait n symboles à encoder. Une première étape est de faire un encodage appelé *one-hot* dans $\mathbb{R}^n$: chaque symbole se voit attribuer un index (on obtient ainsi un dictionnaire des symboles) et est représenté dans un espace continu par un vecteur de dimension n nul partout sauf dans la dimension de son index mise à 1. La deuxième étape consiste à apprendre une représentation dans un deuxième espace de dimension n' (avec $n' < n$ en général - pour le cas des caractères, il n'y aura pas une grande différence) par une projection linéaire. Cet apprentissage peut se faire indépendamment de la tâche de génération ou plus classiquement en même temps que l'apprentissage du réseau général (ce que vous ferez pendant ce TP). Cette représentation peut être ensuite utilisée comme entrée du réseau.

Génération Une fois le réseau appris, la génération se fait (soit à partir d'un début de séquence, soit à partir d'un état initial vide) en choisissant le symbole le plus probable dans la distribution multinomiale décodée à chaque pas de temps. Ce symbole est ensuite considéré comme entrée au pas de temps suivant et la génération se poursuit itérativement. Une autre possibilité est d'échantillonner suivant la multinomiale pour obtenir plusieurs échantillons.

Dans ce TP, nous nous limiterons à la génération de séquences de **taille fixe** que l'on déterminera à l'avance.

Question 4

Le deuxième jeu de données fourni est un ensemble de discours pré-électoraux de Trump. Le but est d'apprendre un RNN qui permet de engendrer un discours à la Trump.

Vous pouvez utiliser le bout de code suivant pour pré-traiter le texte (enlève les caractères accentués et non usuels) et transformer en liste d'index une chaîne de caractères :

```
LETTRES = string.ascii_letters + string.punctuation+string.digits+' '
id2lettre = dict(zip(range(1,len(LETTRES)+1),LETTRES))
```

```
id2lettre[0]='' ##NULL CHARACTER
lettre2id = dict(zip(id2lettre.values(),id2lettre.keys()))

def normalize(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s) if  c in LETTRES)
def string2code(s):
    return torch.tensor([lettre2id[c] for c in normalize(s)])
def code2string(t):
    if type(t) !=list:
        t = t.tolist()
    return ''.join(id2lettre[i] for i in t)
```