04/02/2017

[RNN Hiérarchiques multi-échelles]

¤¤¤ *Rapport final* ¤¤¤

Projet Apprentissage Statistique – M2 DAC, UPMC

Li Ke - Rani Husseini

Table des matières

Contexte :

L'un des problèmes les plus étudiés en matière de réseaux de neurones est la compréhension par la machine de la structure hiérarchique et temporelle des données et de réussir à en tirer une représentation sur plusieurs niveaux d'abstraction. Ce qui permettrait au réseau de neurones en question de pouvoir profiter de la généralisation envers des exemples non vus, du partage de la connaissance apprise entre les différentes taches ou la découverte de nouveaux facteurs de variation entremêlés.

Récemment les CNNs profonds ont prouvées avoir une grande capacité à apprendre la représentation hiérarchique des données spatiales et les RNNs ont menés vers de bonnes avancées dans la modélisation de données temporelles. Inversement l'apprentissage d'une représentation à la fois temporelle et hiérarchique est longtemps resté un défi pour les RNNs.

Schmidhuber (1992), El Hihi et Bengio (1995) et Koutník et al. (2014) ont proposés une approche prometteuse pour modéliser de telles représentations appelée les RNNs multi-échelles (multiscale RNN ou M-RNN). Cette approche est basée sur le fait que l'abstraction dite « haut niveau » change lentement dans le temps par rapport à l'abstraction dite « bas niveau » dont les caractéristiques changent très rapidement et sont sensibles au temps local précis. D'où les M-RNN groupe des couches cachées en multiples modules d'échelles de temps différente. Cette approche apporte également multiples avantages par rapport au RNNs classique qui sont détaillés dans l'article (efficacité de calcul obtenue, allocation de ressources flexible, le partage d'information en aval vers les sous-tâches ou sou-niveaux d'abstraction des données).

Plusieurs implémentations de RNNs multi-échelles ont été proposées. La plus populaire est l'approche consistant à considérer les échelles de temps (ou délais) comme des hyper paramètres à fixer plutôt que de les traiter comme des variables dynamiques qui peuvent être apprises. Mais en prenant en compte le fait que les données temporelles sont généralement non-stationnaires et que beaucoup d'entités d'abstraction (comme les mots et les phrases par exemple) ont des tailles variables, les auteurs de l'article Hierarchical Multiscale Recurrent Neural Networks (Junyoung Chung, Sungjin Ahn et Yoshua Bengio de l'Université de Montréal) préconisent de construire un modèle de RNN qui adapte ses échelles de temps de façon dynamique en fonction des entités données en entrée (de tailles différentes).

Ils proposent donc un nouveau modèle de RNNs multi-échelles capable d'apprendre la structure hiérarchique multi-échelle des données temporelles sans avoir d'information spécifique sur les limites temporelles. Ce modèle, appelé RNN Hiérarchique Multi-échelle (HM-RNN), ne fixe de taux de mises à jour mais détermine de façon adaptative les délais propres de mises à jour correspondants aux différents niveaux d'abstraction des couches.

Le modèle HM-RNN :

Il a été observé que ce modèle apprend des délais précis pour les couches d'abstractions de « bas niveau » et des délais plus grossiers pour les couches d'abstractions de « haut niveau ». Pour ce faire, il faut introduire pour chaque niveau d'abstraction ce qui s'appelle un détecteur de limite binaire.

Ce détecteur est allumé à chaque fois qu'une séquence appartenant à un niveau d'abstraction donné est entièrement traitée et est éteint tant que cette séquence est en cours de traitement. Il faut donc lui conférer des états, appelés les états de limite hiérarchique, en implémentant trois opérations : UPDATE, COPY et FLUSH et de choisir l'un d'entre eux à chaque pas de temps.

Problématiques et contributions de l'article :

Dans l'article cité précédemment tente de répondre à trois problématiques :

- Implémenter un modèle de RNN capable d'apprendre la structure hiérarchique latente d'une séquence sans avoir à priori des informations implicites sur ses limites.
- Montrer que l'estimateur direct (straight-through) est une bonne façon de d'apprendre un modèle contenant des variables discrètes.
- Proposer l'astuce « slope annealing trick » pour améliorer la procédure d'apprentissage basée sur l'estimateur direct.

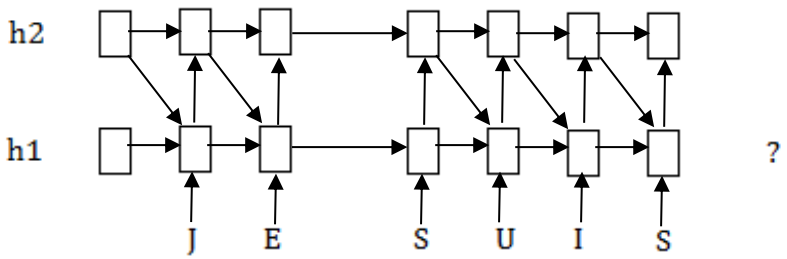
Modèle choisi :

Après concertation avec M. Denoyer, nous avons opté pour un modèle différent de celui utilisé dans l'expérimentation de l'article étudié. Mais qui remplit le même objectif.

Notre modèle s'apparente au modèle de GRUs vu en TME pour la génération de texte, sauf que nous n'apprenons plus uniquement sur les caractères mais nous lui ajoutons une couche supplémentaire qui permet d'obtenir une représentation des mots.

Notre objectif était double, apprendre la structure du texte et prédire correctement l'emplacement des espaces (i.e. la fin des mots).

Architecture du modèle :



Au premier niveau (niveau « caractères ») :

$$h^1_{t+1} = tanh\ (\ z_t * V^1 * h^2_t + (1-z_t) * V^2 * h^1_t\ )$$

L'état du module dépend de son état au pas précédent et de l'état précédent du module d'abstraction supérieure (niveau « mot »). Or nous souhaitons que la représentation du niveau « mot » ne soit récupérée qu'à la fin de chaque mot. On ajoute donc le facteur $\mathbf{z_t}$, $\mathbf{z_t}$ étant notre prédiction (prochain caractère) et prenant comme valeur 1 quand on prédit un espace, 0 sinon. Donc lorsque $\mathbf{z_t}$=0, $\mathbf{h^1_{t+1}}$ ne dépendra que de $\mathbf{h1^t}$ et lorsque $\mathbf{z_t}$=1 il dépendra de la représentation du niveau « mot » (fonction UPDATE). Cette couche transmet ensuite une représentation niveau « caractère » au pas suivant.

Au second niveau (niveau « mot ») :

$$h^2_t = z_t * tanh\ (\ W^1 * h^2_{t-1} + W^2 * h^1_t) + (1-z_t) * h^2_{t-1}$$

L'état du module dépend de son état précédent et de l'état courant du module « caractère ». Ce module peut effectuer deux fonctions : UPDATE lorsque nous arrivons à la fin d'un mot afin que la représentation des mots précédents soit transmise aux prochains mots. COPY lorsque $\mathbf{z_t}$ = 0, qui est une simple copy de l'état actuel à l'état prochain.

Prédiction :

$$z_t = Sigmoid\ (\ T*f_\theta(h^1_t)\ ) = Sigmoid\ (\ T*\theta*h^1_t\ )$$

Il s'agit de notre prédiction (du prochain caractère dans la séquence. Dans notre cas on souhaite vérifier si on atteint la fin d'un mot ou pas. $\mathbf{z_t}$ sera donc 1 s'il s'agit d'un espace et 0 sinon. La Sigmoïde est justement appliquée pour avoir des valeurs autour de 1 ou 0. Au fur et à mesure de l'apprentissage la valeur de $\mathbf{z_t}$ aura tendance à converger à un valeur moyennes entre 0 et 1. Pour cela on rajoute un facteur T qu'on augmente à chaque itération pour contracter la sigmoïde.

Plan expérimental :

1. Problème :

Nous souhaitons construire un modèle capable d'apprendre la structure hiérarchique d'un texte à deux échelles d'abstraction (niveau caractère et niveau mot). Une fois appris, ce modèle servira à prédire la suite d'une séquence de caractères ou de mots.

2. Données :

Nous allons utiliser comme base d'apprentissage le roman « Le petit prince » en anglais. Les données seront préprocéssées d'une façon particulière que nous expliquerons dans la partie suivante.

3. Travail effectué :

Notre projet est divisé en 2 étapes : l'apprentissage sur les données et le test du système.

Dans la partie apprentissage nous avons 2 modules. Un module d'entrée qui permet de récupérer les données sous format texte, d'en créer des embeddings sous format torch (data.t7) et de construire un vocabulaire (vocab.t7). Et un module d'apprentissage (train.lua) dans lequel on entraîne notre modèle explicité plus haut. Une fois l'apprentissage effectué, le modèle (ses paramètres) sont sauvegardée sous format t7 également (model.t7).

Enfin le module de test permet de générer la suite d'une séquence de texte qui lui est donnée et évaluer le système.

- ### Module d'entrée :

data/CharLMMinibatchLoader.lua & Embedding.lua : permet de créer des embeddings à partir de notre fichier text. On obtient un fichier de correspondance caratère ↔ embedding (vocab.t7) et un fichier data.t7 contenant notre texte traduit en embeddings.

- ### Module d'apprentissage (HM-RNN) :

HMRNNL1.lua : contient notre module de niveau « caractère » (h1)

```
function HMRNNL1.hmrnnl1(opt)
    local x = nn.Identity()()
    local prev_h1 = nn.Identity()()
    local prev_h2 = nn.Identity()()
    local prev_z = nn.Identity()()
    local one = nn.Identity()()

    local lin_prev_h1 = nn.Linear(opt.rnn_size, opt.rnn_size)(prev_h1)
    local lin_prev_h2 = nn.Linear(opt.rnn_size, opt.rnn_size)(prev_h2)
    local lin_x = nn.Linear(opt.rnn_size, opt.rnn_size)(x)

    local sub = nn.CSubTable()({one,prev_z})

    local mul1 = nn.CMulTable()({sub, lin_prev_h1})
    local mul2 = nn.CMulTable()({prev_z, lin_prev_h2})

    local add1 = nn.CAddTable()({mul1, mul2})
    local add2 = nn.CAddTable()({add1, lin_x})
    local next_h1 = nn.Tanh()(add2)
    return nn.gModule({x, prev_h1, prev_h2, prev_z, one}, {next_h1})
end
```

HMRNNL2.lua : contient notre module de niveau « mot » (h2)

```
function HMRNNL2.hmrnnl2(opt)
    local prev_h1 = nn.Identity()()
    local prev_h2 = nn.Identity()()
    local prev_z = nn.Identity()()
    local one = nn.Identity()()

    local lin_prev_h1 = nn.Linear(opt.rnn_size, opt.rnn_size)(prev_h1)
    local lin_prev_h2 = nn.Linear(opt.rnn_size, opt.rnn_size)(prev_h2)
    local add1 = nn.CAddTable()({lin_prev_h1, lin_prev_h2})
    local activ = nn.Tanh()(add1)

    local sub = nn.CSubTable()({one,prev_z})

    local mul1 = nn.CMulTable()({prev_z, activ})
    local mul2 = nn.CMulTable()({sub, prev_h2})

    local next_h2 = nn.CAddTable()({mul1, mul2})

    return nn.gModule({prev_h1, prev_h2, prev_z, one}, {next_h2})
end
```

Z.lua : contient notre module de sortie (de prédiction).

```
function Z.z(opt)
    local next_h1 = nn.Identity()()
    local T = nn.Identity()()
    local lin_next_h1 = nn.Linear(opt.rnn_size, opt.rnn_size)(next_h1)
    local mul = nn.CMulTable()({T, lin_next_h1})
    local z = nn.Sigmoid()(mul)
    return nn.gModule({next_h1, T}, {z})
end
```

train.lua : algorithme d'apprentissage.

Nous commençons par récupérer et parser les paramètres d'entrée du sytème d'apprentissage, nous récupérons les données (en batchs). Puis nous créons un prototype de notre modèle entier. Nous fixons l'état initial et nous clonons.

Dans la fonction feval : nous récupérons nos batchs et appliquons les passes forward et backward, enfin nous retournons les valeurs de loss et des gradients.

- ## Module de test :

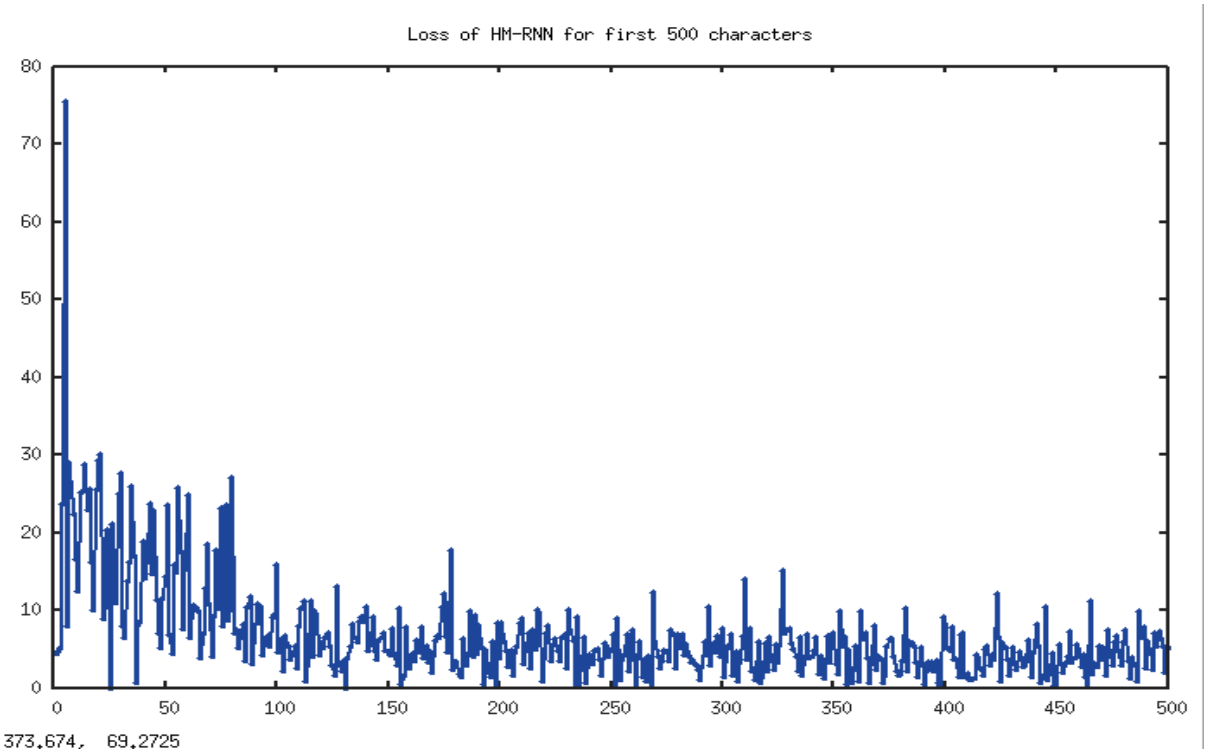
test.lua : génération de la suite d'un séquence (une phrase).

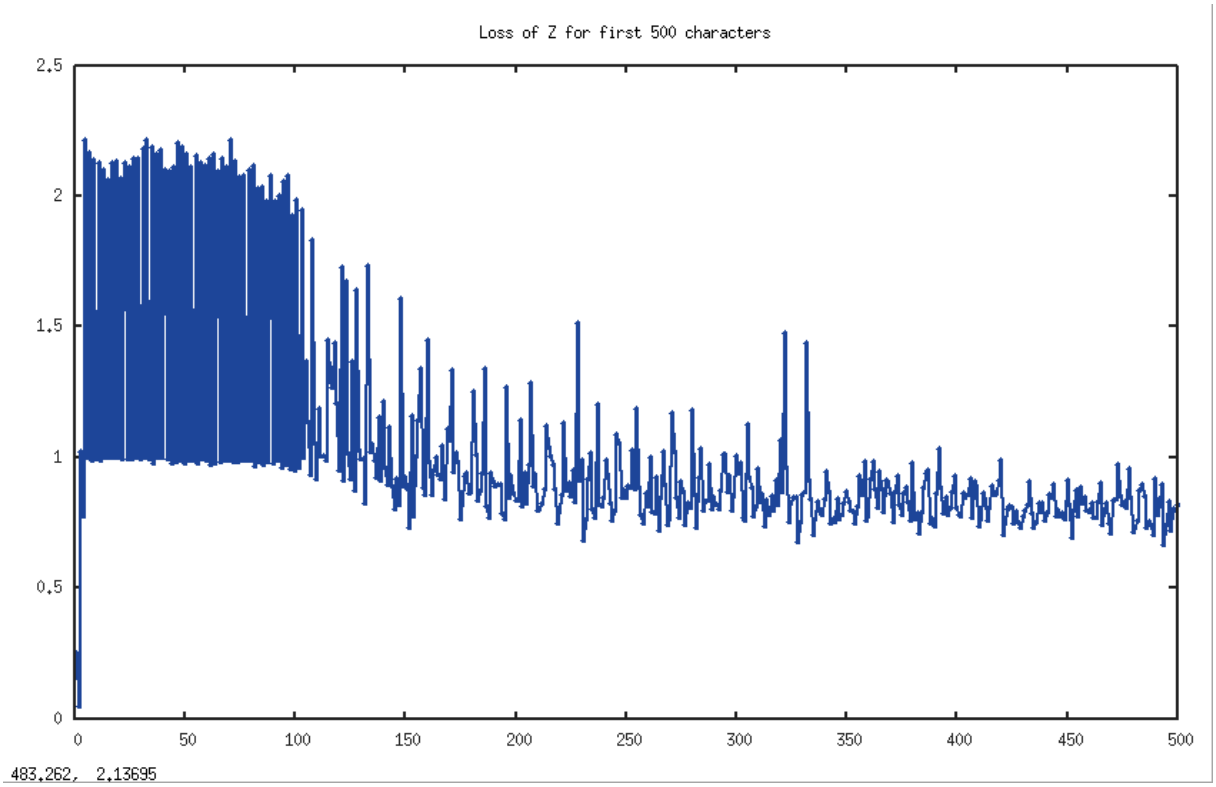
Nous commençons par récupérer les paramètres de notre système que nous parsons. Nous chargeons le modèle HMRNN (appris) obtenus après notre étape de train. Nous récupérons la séquence de lettres donnée en test (par l'utilisateur) et en créons des batches d'une lettre chacun. Puis nous échantillonnons et appliquons la génération de la suite de la séquence.

4. Résultats :

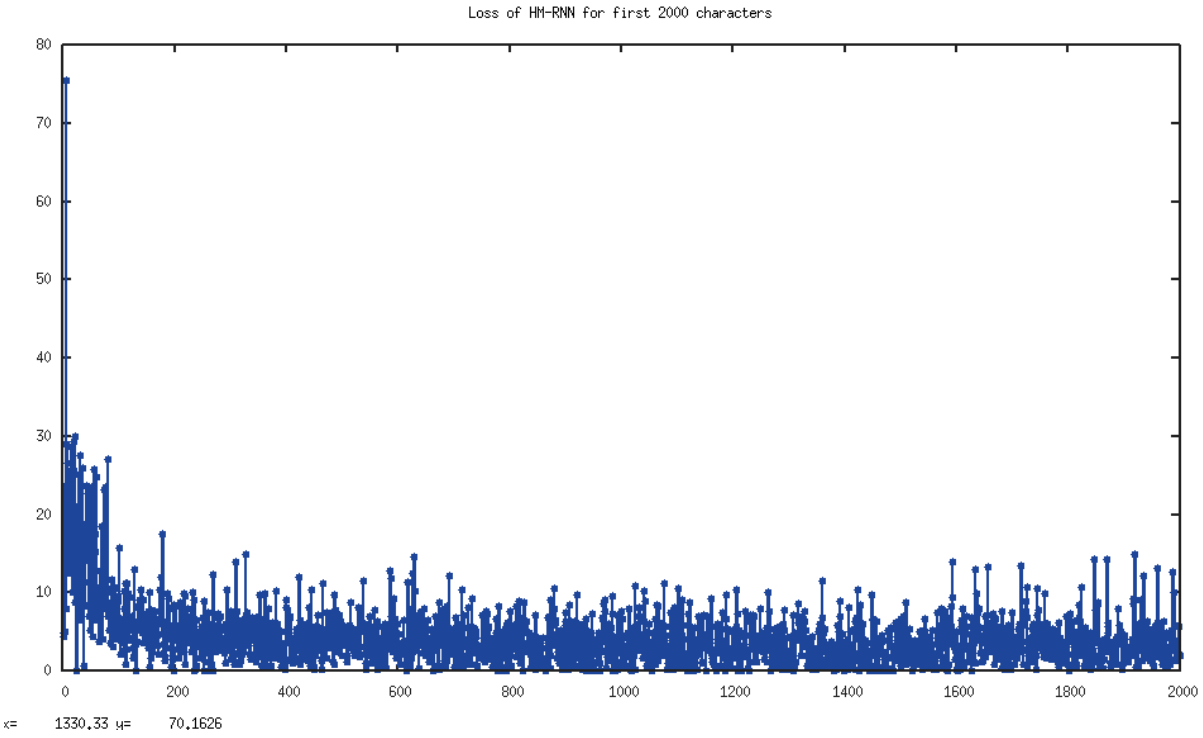
Pour s'assurer du fonctionnement de notre modèle, nous avons sorti les graphes de loss pour le modèle Z et le modèle HM-RNN respectivement.

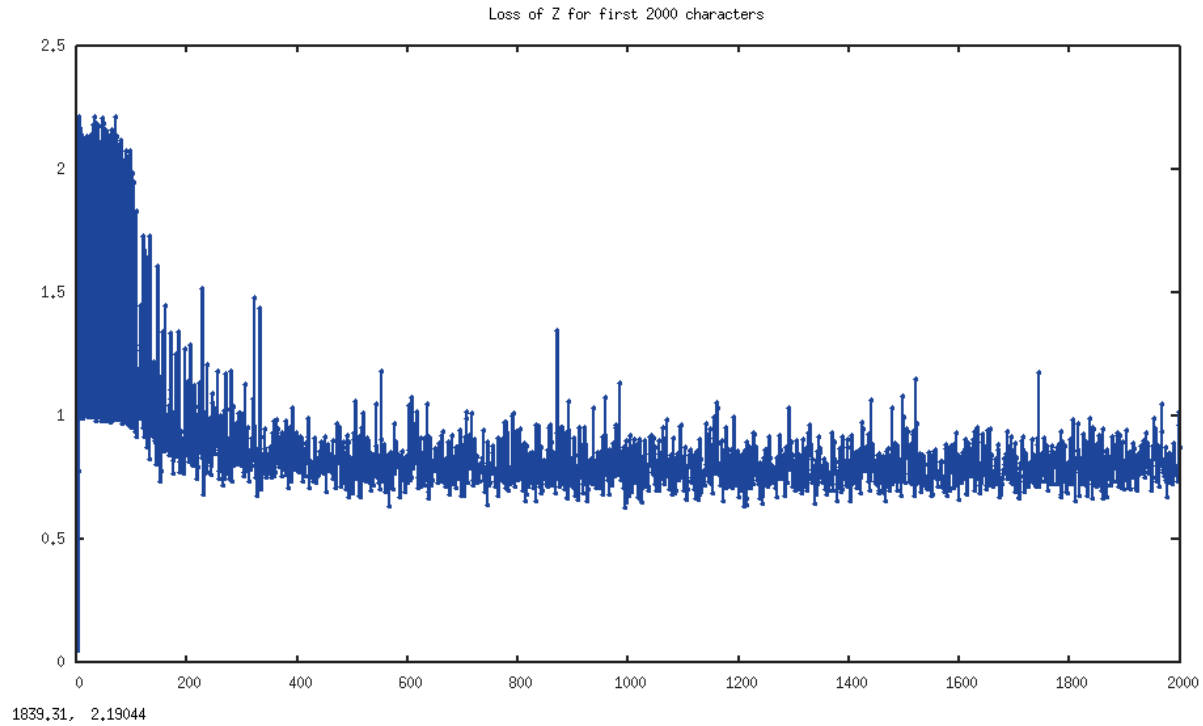
Les graphes de loss pour l'apprentissage sur 500 caractères :





Les graphes de loss pour l'apprentissage sur 2000 caractères :





Ensuite, pour tester la prédiction de séquence, on donne à notre module de test la séquence « I » qu'il doit compléter.

Test avec un modèle qui a appris sur 500 caractères :

```
root@li:/home/like/M2/HM-RNN# th test.lua -text "I"
s-feasqfasopvarhe of qghoa qeasoeaicld-feasoithesi
```

Le modèle peut prédire un mot correct mais il ne peut pas bien séparer cette séquence.

Test avec un modèle qui a appris sur 8000 caractères :

```
root@li:/home/like/M2/HM-RNN# th test.lua -text "I"
WCO."TGis, ly mile filof has lom la ithingolld fol
```

Le modèle peut prédire un mot correct et quelques signes de ponctuation.

Test avec un modèle qui a appris sur le roman en entier :

```
root@li:/home/like/M2/AS/projet/HM-RNN# th test.lua -text "I"
nd ellly, hevengh nes,"
"And on. DOnd end fot her
```

Le modèle peut prédire un peu plus de mots correctement et quelques signes de ponctuation. Mais puisque le texte à apprendre n'est pas long, la prédiction dans notre cas ne marche pas très bien.

Pour prédire une séquence parfaite, il faut apprendre beaucoup plus de texte. Mais comme notre modèle HM-RNN fait une détection sur chaque caractère entrant, cela prend beaucoup de temps (4 heures pour apprendre le roman en entier) en comparant avec le modèle RNN.