POLYTECH SORBONNE

SCIENCES SORBONNE UNIVERSITÉ

Devoirs maison

réalisé par
Arthur Zucker

Apprentissage statistique et Machine Learning

encadré par

Patrick Gallinari

Mathématiques Appliquées et Informatique

3e année de cycle ingénieur

7 décembre 2020

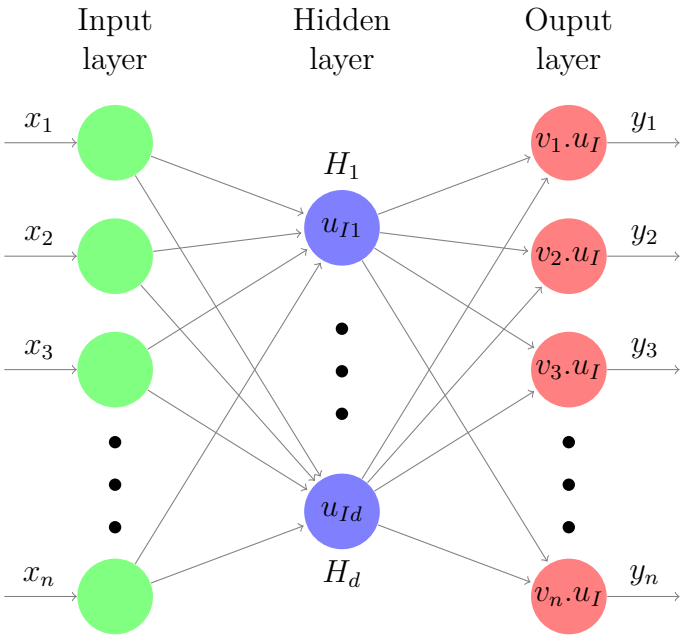
Paris, France

Introduction

La forme du réseau étudié est celle d'un **encoder**, très utilisé par exemple pour réduire la dimension des caractéristiques ou "features" donné en entrée. Je me suis amusé à coder le réseau proposé, afin de voir quel performances on pouvait obtenir. Le code est fourni en annexe. On utilisant un corpus de texte provenant d'un livre par exemple, de manière instinctive, on se doute que le réseau va donner beaucoup d'importance à des mots qui apparaissent plusieurs fois avec un certain contexte. Ainsi, si le mots "de" peut être suivis de toutes sortes d'autres mots, il est très souvent suivit de "le" ou "la" par exemple, et ainsi les y_{le} et y_{la} seront élevés proportionnellement à leur apparition dans le corpus.

Le réseau est composé d'une input layer de dimension n. Sachant d'avance que chacun des mots du dictionnaire est codé en utilisant le "one-hot" encoding, on sait qu'un seul des neurones d'entrée sera activé pour chacun des mots qui seront fournis en entrée. Tous les neurones de l'input layer sont connectés à la couche cachée, il en va de même pour les neurones de la couches cachée, qui sont tous connectés à la couche de sortie. On a donc une architecture de FCNN ou Fully Connected Neural Network.

On peut représenter ce réseau de la manière suivante.



On a en sortie les $y_k = \frac{\exp(v_k.u_I)}{\Sigma_{j=1}^{n} \exp(v_j.u_I)}$

Question 1

On sait que :

$$\log p(w_O|w_I) = v_O.u_I - \log(\Sigma_{s=1}^{n} \exp(v_s.u_I)) \tag{1}$$

Ainsi, maximiser la log vraisemblance revient à :

- maximiser $-\log(\Sigma_{s=1}^{n} \exp(v_s.u_I))$ et donc à minimiser $\log(\Sigma_{s=1}^{n} \exp(v_s.u_I))$
- maximiser $v_O.u_I$

Maximiser la log vraisemblance reviens aussi à maximiser $p(w_O|w_I)$.

Les normes des vecteurs v_O et u_I vont donc augmenter puisque maximiser $v_O.u_I$ reviens donc à maximiser les produits intermédiaires, et ainsi les valeurs en absolue des coefficients de v_O et u_I.

On utilise la log vraisemblance parce que l'expression obtenu est facile à maximiser : on a une somme et non pas des produit et des quotients.

Maximiser la vraisemblance reviens à essayer d'augmenter la probabilité que, en donnant en entrée un mot au réseau, celui-ci nous rendent un mot qui vient souvent après l'input. Par exemple, le réseau est censé apprendre à renvoyer "chien" ou "chat" lorsqu'on lui donne en entrée "le".

Question 2

Il est évident que les représentations de deux mots w_I et w_J tels que leur "label" et donc leur contexte est w_O aurons une représentation $u_I \approx u_J$ puisque qu'on aura :

$$\text{Pour I} : y_O = \frac{\exp(v_O.u_I)}{\Sigma_{j=1}^{n} \exp(v_j.u_I)}$$

$$\text{Pour J} : y_O = \frac{\exp(v_O.u_J)}{\Sigma_{j=1}^{n} \exp(v_j.u_J)}$$

$$\text{D'où} : v_O.u_I \approx v_O.u_J$$

On en déduit que le représentation apprises des deux mots seront similaires.

Question 3

1 : Expression du log de la vraisemblance

D'après l'énoncé on sait que

$$p(w_I|w_O) = \frac{\exp(v_O.u_I)}{\Sigma_{j=1}^{n} \exp(v_j.u_I)} \tag{2}$$

On en déduit que :

$$\begin{aligned} e(w_I, w_O) &= \log p(w_O|w_I) \\ &= v_O.u_I - \log(\Sigma_{s=1}^n \exp(v_s.u_I)) \\ &= a_O - \log(\Sigma_{s=1}^n \exp(a_s)) \\ &= \Sigma_{t=1}^n v_O b_t - \log(\Sigma_{s=1}^n \exp(\Sigma_{t=1}^n v_{st} b_t)) \end{aligned}$$

Avec $a_s = v_s.u_I$ et $b_t = \Sigma_{s=1}^n u_{ts} x_{Is}$. On a utilisé le fait que :

$$\begin{aligned} v_s.u_I = v_s.(U.x_I) &= v_s. \begin{pmatrix} \Sigma_{s=1}^n u_{1s} x_{Is} \\ \Sigma_{s=1}^n u_{2s} x_{Is} \\ \vdots \\ \Sigma_{s=1}^n u_{ns} x_{Is} \end{pmatrix} \\ &= v_s. \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\ &= \begin{pmatrix} v_{s1} v_{s2} \dots v_{sn} \end{pmatrix} . \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\ &= \Sigma_{t=1}^n v_{st} b_t \end{aligned}$$

Ici s et t sont des variables muettes, on peut les remplacer par n'importe quelles autres variables, mais il n'est pas très judicieux d'utiliser i, j puisque cela pourrait porter à confusion lors du calcul des dérivés. Ainsi, lorsque l'on va dériver par rapport à u_{ij} il s'agira en fait de fixer u_{st} en $s = i, t = j$ comme étant la variable de dérivation. Cela nous permettra d'y voir plus clair.

2 : "The chain rule"

En utilisant les règle de dérivation dite de "chaine", nous allons calculer les différentes dérivées partielles de e.

3.2.a On a $\frac{\partial e}{\partial v_{ij}} = \frac{\partial e}{\partial a_i} . \frac{\partial a_i}{\partial v_{ij}} = \frac{-exp(a_i)}{\Sigma_{s=1}^n \exp(a_s)} \times b_j$. Obtenu en utilisant 3 et 4

$$\frac{\partial e}{\partial a_i} = \frac{\partial (a_O - \log(\Sigma_{s=1}^n \exp(a_s)))}{\partial a_i} \tag{3}$$

On remarque que quand $s = i$, la dérivée de $\Sigma_{s=1}^{n} \exp(a_s))$ vaut $a_i exp(a_i)$. En utilisant les règle de dérivation de fonctions composées, il vient :

$$\frac{\partial e}{\partial a_i} = \frac{-\exp(a_i)}{\Sigma_{s=1}^{n} \exp(a_s)}$$

. Et d'un autre coté :

$$\frac{\partial a_i}{\partial v_{ij}} = b_j \text{ sachant que } a_i = \Sigma_{t=1}^{n} v_{it} b_t \tag{4}$$

On évalue la dérivée dont le seul terme non constant est en $t = j$.
On en déduit que :

$$\frac{\partial e}{\partial v_{ij}} = -y_i b_j \tag{5}$$

3.2.b Calculons maintenant $\frac{\partial e}{\partial u_{ij}}$. En utilisant les notations précédemment introduites :

$$\begin{aligned}
\frac{\partial e}{\partial u_{ij}} &= \frac{\partial e}{\partial b_i} . \frac{\partial b_i}{\partial u_{ij}} \\
&= [\Sigma_{k=1}^{n} \frac{\partial e}{\partial a_k} \frac{\partial a_k}{\partial b_i}] \times x_{Ij} \\
&= [\frac{-\Sigma_{k=1}^{n} v_{ki} . exp(a_k)}{\Sigma_{s=1}^{n} exp(a_s)}] \times x_{Ij}
\end{aligned}$$

On a utilisé 3, il vient

$$\frac{\partial e}{\partial u_{ij}} = -x_{Ij} \Sigma_{k=1}^{n} y_k \times v_{ki} \tag{6}$$

Algorithme de gradient stochastique

En utilisant 5 et 6, en posant α le "learning rate", D le dictionnaire, l'algorithme de mise à jour des poids est le suivant :

Algorithm 1 Descente de gradient stochastique

1: **procedure** SGD(D,U,V,α)
2: $(w_I, w_O) \leftarrow$ un mot choisi au hasard et son label $\in$ **D**
3: Effectuer une forward propagation en calculant les y_k $\forall k$
4: Les calculs de $\frac{\partial e}{\partial v_{ij}}$ et $\frac{\partial e}{\partial u_{ij}}$ se font avec 5 et 6
5: **for** i=1,n **do**
6: **for** j=1,d **do**
7: $v_{ji} \leftarrow v_{ji} + \alpha \times \frac{\partial e}{\partial v_{ji}}$
8: $u_{ij} \leftarrow u_{ij} + \alpha \times \frac{\partial e}{\partial u_{ij}}$

On effectue ensuite cet algorithme sur tous les mots du dictionnaire.

Question 4 :

Complexité de l'échantillonnage de manière naïve

Le calcul de $p(w_O|w_I)$ se fait en $O(n^2)$ en utilisant 2, et en sachant que le produit matrice vecteur se fait en $O(n^2)$, produit scalaire en $O(n)$ et que l'on doit effectuer n produits scalaires entre v_s et u_I. Et que u_I s'obtient en $O(n^2)$ opérations.

Complexité de la mise à jour

La complexité de la mise à jour des poids est en $O(n^3)$ en utilisant l'algorithme 1 introduit ci-dessus. On doit tout d'abord calculer les y_k ($O(n^2)$). Ensuite, on a deux boucles imbriquées, dans lesquelles le calcul de $\frac{\partial e}{\partial u_{ij}}$ nécessite $O(n)$ si on l'effectue en dehors de la boucle. Sinon, et comme dans cette question on demande à la manière naïve, on doit effectuer n fois ce calcul et on obtient une complexité en $O(n^3)$. De plus, si on ne stock pas les y_k, et qu'on les recalcules pour chaque $\frac{\partial e}{\partial u_{ij}}$, on a une complexité de $O(n^5)$! Des techniques simple pour réduire la complexité sont déjà présenté ici, mais la réponse suivante sera plus structurée.

Alternatives intelligentes

L'alternative est d'utiliser la **tokenization**, on donne en entrée l'indice du mot dans le dictionnaire, et on cherche à avoir en sortie l'indice du mot du dictionnaire cible. Ainsi, on peut obtenir u_I sans avoir à effectuer un produit matrice vecteur, il suffit de donner directement la $I^{\text{ème}}$ colonne de u. Ensuite, le calcul des $y_k \forall k$ peut se faire en $O(n^2)$. Pour cela il suffit de calculer les $\exp(v_j.u_I)$, de les stocker et d'ensuite les sommer. Ensuite, $\forall i, \Sigma_{k=1}^{n} - y_k \times v_{ki}$, enfin, on actualise les colonnes simultanément en multipliant pour V par x_{Ij} et en multipliant y_i par b_j pour U

Programmation du réseau

Le code suivant permet en python d'effectuer l'entraînement du réseau. Je n'ai pas eu le temps d'optimiser les paramètres, mais j'ai implémenté le mini-batch SGD.

```
import nltk
import numpy as np
from sklearn.preprocessing import OneHotEncoder
import nltk
from nltk.corpus import words
import pandas as pd
import nltk
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, TensorDataset
from torchvision import transforms, utils


class Dataset(Dataset):
    def __init__(self, X):
        self.X = X[:,:-1]
        self.y = X[:,1:]
    def __len__(self):
        return (self.X).shape[1]
    def __getitem__(self, idx):
        return self.X[:,idx], self.y[:,idx]

class Encoder(nn.Module):
        def __init__(self, input_size, d,batch_size =5000):
                super(Encoder, self).__init__()
                self.input_size = input_size
                self.d = d
                self.fc1 = nn.Linear(in_features=self.input_size,
                    out_features=d)
                self.fc2 = nn.Linear(in_features=self.d,
                    out_features=self.input_size)
                self.soft = nn.LogSoftmax(dim=-1)

        def forward(self, x):
                y1 = self.fc1(x)
                y2 = self.soft(self.fc2(y1))
                return y2


use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
print('Device: ', device)

word_list = words.words()
emma = nltk.corpus.gutenberg.words('austen-emma.txt')
```

Listing 1 – Code de l'encoder

```
di = {}
i = 0
for word in emma:
        if word not in di:
                di[word] = i
                i += 1

def encode(word, di):
        size = len(di)
        tab = torch.zeros(size)
        tab[di[word]] = 1
        return tab

batch_size = 150
n = 1000*batch_size
X = torch.Tensor(len(di), n)
for i in range(n):
        X[:,i] = encode(emma[i], di)

print(X.shape)

rate = 0.8
train_rate = int(0.8*X.shape[1])
model = (Encoder(X.shape[0], 150)).to(device)
loss = nn.NLLLoss().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
i = 0
```

Listing 2 – Création du dictionnaire

```
ds = Dataset(X)
ds = DataLoader(ds, batch_size=batch_size, shuffle=True,pin_memory=True)

num_epochs = 20
losses = []
for e in range(num_epochs):
  #model.train()
  for ix, (_x, _y) in enumerate(ds):
        _x = _x.to(device)
        _y = _y.to(device)
        #========forward pass=====================================
        yhat = model(_x)

        if (_y.shape[0]==batch_size) :
                target = torch.Tensor([torch.where(_y[i] == 1)[0]
                        for i in range(batch_size)]).to(device)
                l = loss(yhat, target.long())


                #=======backward pass=====================================
                optimizer.zero_grad() # zero the gradients on each
                #pass before the update
                l.backward() # backpropagate the loss through the model
                optimizer.step() # update the gradients w.r.t the loss

                losses.append(l.item())




        else:
        print(_y.shape)
  print("[{}/{}], loss: {} ".format(e,num_epochs, np.round(l.item(), 3)))
```

Listing 3 – Entrainement