PSTALN_TP06_langmodel

January 3, 2022

1 Language models

- Notebook créé par Benoit Favre
- Adapté par Carlos Ramisch

La plupart de ces notebooks peuvent tourner sans acceleration GPU. Vous pouvez utiliser Google Colab ou installer jupyter-notebook sur votre ordinateur. Pytorch est déjà installé sur Colab, par contre il faut suivre https://pytorch.org/get-started/locally/ en local.

[ ]:
```
## example de deploiement local
# virtualenv -ppython3.8 pstaln-env
# source pstaln-env/bin/activate
# pip3 install torch==1.10.0+cpu -f https://download.pytorch.org/whl/cpu/
 ↪torch_stable.html
# pip3 install matplotlib ipykernel
# python3 -m ipykernel install --user --name=pstaln-env
# jupyter-notebook
## puis sélectionner le noyeau pstaln-evn
```

La un **modèle de langage** est un outil capable d'apprendre la distribution de probabilité d'un mot étant donné le contexte (ou l'historique) des mots qui l'entourent (ou précèdent). Ici, nous allons créer un modèle de langage sur des caractères pour apprendre à générer des titres de films de science fiction. Cela nous permet de garder un vocabulaire de taille limitée lors de l'utilisation d'un softmax sur la sortie d'un classifieur qui prédit le prochain caractère.

Le jeu de données provient d'IMDB qui permet d'accèder à de nombreuses infos sur les films. Ces données sont fournies en téléchargement libre (http://www.imdb.com/interfaces/). Le fichier `movies-sf.txt` contient des noms de films suivis de leur année de sortie entre parenthèses extraits à partir de la base de données IMDB à l'aide de la commande `awk` montrée en commentaire ci-dessous.

[2]:
```
%%bash
# commandes utilisées pour télécharger et pré-traiter les données
# wget https://datasets.imdbws.com/title.basics.tsv.gz
# zcat title.basics.tsv.gz | awk -F"\t" '$2=="movie" && $5==0 && /Sci-Fi/ && $6!
 ↪="\\N"{print $3" ("$6")"}' | iconv -f utf8 -t ascii//TRANSLIT | sort -u |␣
 ↪shuf > movies-sf.txt
if [ ! -f movies-sf.txt ]; then
    wget -q https://pageperso.lis-lab.fr/carlos.ramisch/download_files/pstaln/
 ↪movies-sf.txt
```

```
fi
head movies-sf.txt
```

```
Passengers (2016)
Stealth (2005)
Utterance (1997)
Homunculus, 6. Teil - Das Ende des Homunculus (1917)
Framework (2009)
Redshift (2013)
Jupiter 2023 (2018)
Fuerza maldita (1995)
Horrors of War (2006)
500 MPH Storm (2013)
```

Avant de continuer, nous allons importer les librairies dont nous avons besoin aujourd'hui.

[21]:
```
%matplotlib inline

import collections
from matplotlib import pyplot as plt
import math

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
```

1.0.1 Préparation des données

Nous allons charger les titres caractère par caractère et encoder ces derniers sous forme d'entiers. Le vocabulaire est produit avec un `defaultdict` qui donne un nouvel identifiant à chaque nouveau caractère rencontré. Nous ajoutons deux caractères spéciaux : - le symbole `<eos>` pour le padding - le symbole `<start>` qui indique le début de la séquence

Le problème sera posé comme une tâche de prédiction du caractère suivant étant donné le caractère courant et un état caché. Nous avons donc besoin d'un symbole `<start>` pour prédire le premier caractère. La fin d'un texte sera décrétée lorsque le modèle prédit le symbole `<eos>`.

Nous pouvons tout de suite créer un vocabulaire inversé pour vérifier le contenu des données chargées.

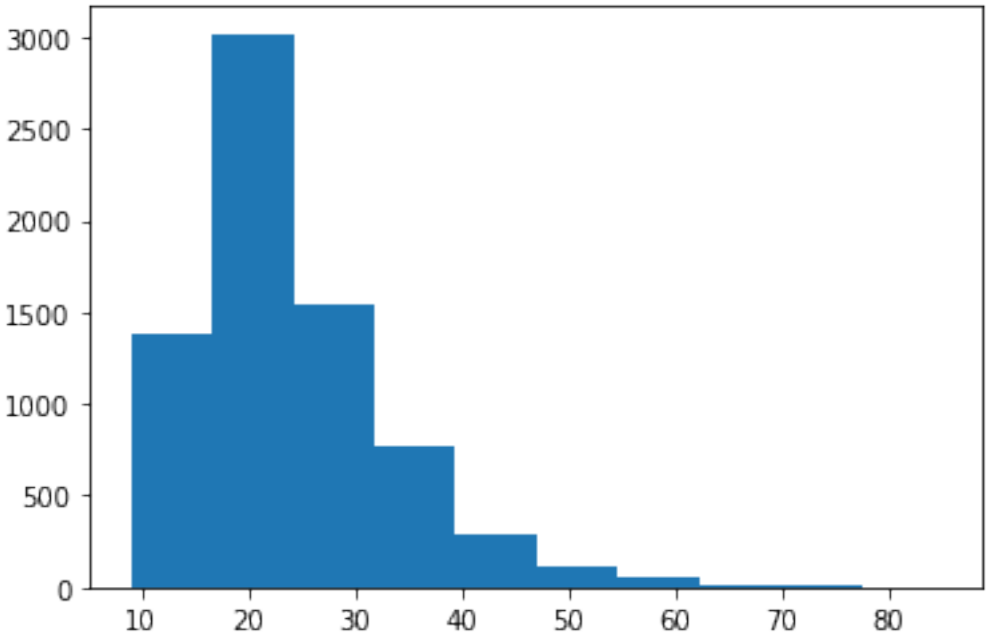
[13]:
```
vocab = collections.defaultdict(lambda: len(vocab))
vocab['<eos>'] = 0
vocab['<start>'] = 1

int_texts = []
with open('movies-sf.txt', encoding="utf-8") as fp:
    for line in fp:
```

```
        int_texts.append([vocab['<start>']] + [vocab[char] for char in line.
↪strip()])

rev_vocab = {y: x for x, y in vocab.items()}

print(rev_vocab)
print(len(int_texts))

print(int_texts[42])
print(''.join([rev_vocab[x] for x in int_texts[42]]))
```

```
{0: '<eos>', 1: '<start>', 2: 'P', 3: 'a', 4: 's', 5: 'e', 6: 'n', 7: 'g', 8:
'r', 9: ' ', 10: '(', 11: '2', 12: '0', 13: '1', 14: '6', 15: ')', 16: 'S', 17:
't', 18: 'l', 19: 'h', 20: '5', 21: 'U', 22: 'c', 23: '9', 24: '7', 25: 'H', 26:
'o', 27: 'm', 28: 'u', 29: ',', 30: '.', 31: 'T', 32: 'i', 33: '-', 34: 'D', 35:
'E', 36: 'd', 37: 'F', 38: 'w', 39: 'k', 40: 'R', 41: 'f', 42: '3', 43: 'J', 44:
'p', 45: '8', 46: 'z', 47: 'W', 48: 'M', 49: 'v', 50: 'A', 51: 'N', 52: '4', 53:
'B', 54: 'V', 55: 'I', 56: 'L', 57: 'G', 58: 'b', 59: 'C', 60: ':', 61: 'X', 62:
'x', 63: 'y', 64: 'O', 65: 'Z', 66: 'j', 67: 'q', 68: 'Y', 69: "'", 70: '?', 71:
'Q', 72: '/', 73: '&', 74: 'K', 75: '!', 76: '=', 77: '_', 78: '+', 79: ';', 80:
'@', 81: '#', 82: '%', 83: '$'}
7205
[1, 31, 19, 5, 9, 2, 32, 6, 39, 9, 59, 19, 32, 67, 28, 32, 17, 3, 4, 9, 10, 13,
23, 45, 24, 15]
<start>The Pink Chiquitas (1987)
```

Afin de bien choisir la longueur maximale sur laquelle le modèle va être entrainé, affichons l'histograme des longueurs de séquences.

[14]:
```
plt.hist([len(text) for text in int_texts])
plt.show()
```



Il semble qu'une longeur maximale de 40 permettra de traiter une bonne partie des titres. Les rares titres dépassant cette longueur maximale seront coupés, comme d'habitude. Nous en profitons pour définir d'autres hyper-paramètres de notre modèle tels que la taille des batches (`8`), des embeddings de caractères (`16`), et de la couche cachée du réseau récurrent (`64`).

[8]:
```
max_len = 40
batch_size = 8
embed_size = 16
hidden_size = 64
```

Nous allons utiliser un réseau de neurones récurrent pour tenir compte de l'historique de caractères. Dans ce type de modèle, un modèle de langage peut être vu comme un modèle de tagging, sauf que l'étiquette à prédire est le caractère suivant.

Nous devons donc agencer les tenseurs de manière à ce que $y_t = x_{t+1}$. Il faut calculer la longueur après la coupure des séquences plus longues que `max_len`, puis créer un tenseur à partir du texte pour x et un tenseur à partir du texte décalé de 1 vers la gauche pour y.

N'oublions pas de vérifier que les données ont la bonne forme.

[15]:
```
X = torch.zeros(len(int_texts), max_len).long()
Y = torch.zeros(len(int_texts), max_len).long()

for i, text in enumerate(int_texts):
    length = min(max_len, len(text) - 1) + 1
    X[i,:length - 1] = torch.LongTensor(text[:length - 1])
    Y[i,:length - 1] = torch.LongTensor(text[1:length])
```

```
print(X[42].tolist())
print(Y[42].tolist())
print([rev_vocab[x] for x in X[42].tolist()][:30])
print([rev_vocab[y] for y in Y[42].tolist()][:30])
```

```
[1, 31, 19, 5, 9, 2, 32, 6, 39, 9, 59, 19, 32, 67, 28, 32, 17, 3, 4, 9, 10, 13,
23, 45, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[31, 19, 5, 9, 2, 32, 6, 39, 9, 59, 19, 32, 67, 28, 32, 17, 3, 4, 9, 10, 13, 23,
45, 24, 15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
['<start>', 'T', 'h', 'e', ' ', 'P', 'i', 'n', 'k', ' ', 'C', 'h', 'i', 'q',
'u', 'i', 't', 'a', 's', ' ', '(', '1', '9', '8', '7', '<eos>', '<eos>',
'<eos>', '<eos>', '<eos>']
['T', 'h', 'e', ' ', 'P', 'i', 'n', 'k', ' ', 'C', 'h', 'i', 'q', 'u', 'i', 't',
'a', 's', ' ', '(', '1', '9', '8', '7', ')', '<eos>', '<eos>', '<eos>', '<eos>',
'<eos>']
```

Nous voyons bien ci-dessus que, pour un caractère donné dans x_i (p.ex.`<start>`) la sortie dans y_i (p.ex. `'T'`) correspond au prochain caractère de l'entrée x_{i+1}.

Nous découpons les données en un ensemble d'entraînement et un ensemble de validation, puis on fait appel aux outils pytorch pour créer des batches mélangés sont utilisés comme d'habitude.

[16]:
```
X_train = X[:6500]
Y_train = Y[:6500]
X_valid = X[6500:]
Y_valid = Y[6500:]

train_set = TensorDataset(X_train, Y_train)
valid_set = TensorDataset(X_valid, Y_valid)

train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=batch_size)
```

1.0.2 Modèle prédictif

Le modèle ressemble beaucoup à un taggeur. La première différence est qu'il ne peut pas être bidirectionnel, puisque la causalité est importante (on va générer des textes caractère par caratère en partant de `<start>`). La seconde différence est que la fonction `forward` va prendre un nouveau paramètre optionnel, l'état caché au temps précédent. Comme le réseau est récurrent, la fonction doit renvoyer non seulmenent les scores générés par le modèle, mais aussi le nouvel état caché après avoir vu la séquence représentée dans `x`. Ceci sera nécessire pour la génération caractère par caractère.

[17]:
```
class LM(nn.Module):
    def __init__(self, vocab):
        super().__init__()
        self.embed = nn.Embedding(len(vocab), embed_size,␣
↪padding_idx=vocab['<eos>'])
```

```
        self.rnn = nn.GRU(embed_size, hidden_size,  bias=False, num_layers=1,␣
↪batch_first=True)
        self.dropout = nn.Dropout(0.3)
        self.decision = nn.Linear(hidden_size, len(vocab))
        ## Optional: tie embedding and decision weights (recommanded for word␣
↪LM)
        # assert embed_size == hidden_size
        # self.decision.weight = self.embed.weight

    def forward(self, x, hidden_0=None):
        embed = self.embed(x)
        output, hidden = self.rnn(embed, hidden_0)
        return self.decision(self.dropout(output)), hidden

model = LM(vocab)
model
```

[17]:
```
LM(
  (embed): Embedding(84, 16, padding_idx=0)
  (rnn): GRU(16, 64, bias=False, batch_first=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (decision): Linear(in_features=64, out_features=84, bias=True)
)
```

Notez que il n'est pas obligatoire de passer un état caché initial `hidden_0`. Le module GRU de la bibliothèque s'occupe d'initialiser les paramètres si l'état caché passé est `None` (valeur par défaut).

Nous pouvons vérifier les dimensions des entrées et sorties du modèle (non entraîné) sur un batch quelconque (p.ex. les deux premiers titres de film). Même si on peut omettre l'état caché initial, on doit récupérer le nouvel état caché renvoyé par `forward`, quitte ne jamais l'utiliser par la suite. Il faut donc systématiquement se souvenir que le modèle renvoie deux résultats (les scores pour chaque caractère et l'état caché) et donc mettre l'état caché dans une variable qui ne sert à rien.

Remarquons que les sorties sont de taille `(batch_size, sequence_length, num_labels)` et l'état caché `(num_layers, batch_size, hidden_size)`. Ici, comme il s'agit d'un modèle de langage et non pas d'un taggeur, `num_labels` est identique à la taille du vocabulaire d'entrée.

[18]:
```
output, hidden = model(X[:2])
print(output.size(), hidden.size())
```

```
torch.Size([2, 40, 84]) torch.Size([1, 2, 64])
```

1.0.3 Évaluation

Il y a quelques différences entre l'évaluation d'un modèle de langage et l'évaluation d'un taggeur. À la place du taux d'étiquettes correctes prédites, nous allons calculer la perplexité du modèle sur les données, comme défini comme la moyenne géométrique des probabilités affectées par le modèle à chaque caractère d'une suite de N caractères d'une suite x donnée pour l'évaluation :

$$PP(x) = P(x)^{-\frac{1}{N}} = \left[\prod_i P(x_i)\right]^{-\frac{1}{N}}$$

où x est une séquence de caractères, $P(x) = \prod_i P(x_i)$ est la probabilité donnée par le modèle à cette séquence, et N est sa longueur. On peut réécrire ce calcul en domaine log pour retrouver la formulation habituelle de la perplexité :

$$PP(x) = exp\left(-\frac{1}{N}\sum_i \log P(x_i)\right)$$

Il se trouve que la fonction de loss `CrossEntropyLoss` renvoie déjà $-\frac{1}{N}\log P(x_i)$ (log-probabilité moyennée sur la longueur du batch N), donc il suffit de calculer l'exponentielle de la loss moyenne pour obtenir la perplexité. Cette perplexité n'est pas masquée pour éliminer le padding, donc elle est influencée par ce dernier. Si on voulait ignorer le padding, on ne pourrait pas profiter de la fonction de loss : il faudrait recalculer le softmax et les logarithmes explicitement.

[23]:
```
def perf(model, loader):
    criterion = nn.CrossEntropyLoss()
    model.eval()
    total_loss = num = 0
    for x, y in loader:
      with torch.no_grad():
        y_scores, _ = model(x)
        loss = criterion(y_scores.view(y.size(0) * y.size(1), -1), y.view(y.
 ↪size(0) * y.size(1)))
        total_loss += loss.item()
        num += len(y)
    return total_loss / num, math.exp(total_loss / num)

perf(model, valid_loader)
```

[23]:
```
(0.1300052286885309, 1.1388343379190957)
```

L'apprentissage est le même que pour le taggeur sauf qu'il faut prendre en compte l'état caché.

[22]:
```
def fit(model, epochs, train_loader, valid_loader):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters())
    for epoch in range(epochs):
        model.train()
        total_loss = num = 0
        for x, y in train_loader:
            optimizer.zero_grad()
            y_scores, _ = model(x)
            loss = criterion(y_scores.view(y.size(0) * y.size(1), -1), y.view(y.
 ↪size(0) * y.size(1)))
```

```
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            num += len(y)
        print(epoch, total_loss / num, *perf(model, valid_loader))

fit(model, 10, train_loader, valid_loader)
```

```
0 0.15247126197814942 0.14502824223633354 1.1560722198718227
1 0.14661014809058262 0.14053412963312567 1.1508883582919969
2 0.14304072733108814 0.13776597351047165 1.1477069250200223
3 0.1408569251390604 0.13581273344391626 1.1454673657798502
4 0.13896740749249092 0.13448289370705896 1.1439450901751997
5 0.13757389782025264 0.13332028879341504 1.1426159067997008
6 0.13657176457918607 0.1325818713675154 1.1417724907378919
7 0.13545437229596652 0.13143955874950328 1.1404689742682226
8 0.13473722082834977 0.13083775533852002 1.1397828426282761
9 0.1336541407291706 0.1300052286885309 1.1388343379190957
```

1.0.4 Utilisation du modèle

Écrivons maintenant une fonction de génération de titres de filmes avec ce modèle. Cette dernère crée un tenseur x contenant le symbole `<start>`, et un état caché à 0. Puis, elle repète l'application du modèle sur x et l'état caché, pour générer un nouvel état caché et un vecteur y_{scores} sur les caractères. On peut alors sélectionner la composante de plus haut score, l'afficher, et mettre à jour x pour qu'il contienne le symbole généré. Il suffit ensuite de boucler jusqu'à la génération de `<eos>`.

Le modèle génère toujours la même séquence de caractères. Il s'agit de la séquence la plus probable étant donné le corpus d'apprentissage.

[31]:
```
def generate_most_probable(model):
    x = torch.zeros((1, 1)).long()
    x[0, 0] = vocab['<start>']
    # size for hidden: (batch, num_layers * num_directions, hidden_size)
    hidden = torch.zeros(1, 1, hidden_size)
    with torch.no_grad():
      for i in range(200): # Longueur maximale pour éviter de boucler si␣
 ↪`<eos>` ne sort pas
        y_scores, hidden = model(x, hidden)
        y_pred = torch.max(y_scores, 2)[1]
        selected = y_pred.data[0, 0].item()
        if selected == vocab['<eos>']:
            break
        print(rev_vocab[selected], end='')
        x[0, 0] = selected
    print()

generate_most_probable(model)
```

```
The Star War (2018)
```

1.1 Exercice 1

Plutôt que de sélectionner le caractère ayant la plus grande probabilité, on peut tirer aléatoirement un caractère dans la distribution de probabilité générée par le softmax. Utilisez `F.softmax` et `torch.multinomial` pour tirer aléatoirement un élément $s \sim softmax(y_{\text{scores}})$ dans la distribution des scores, et l'utiliser comme élément sélectionné à la place de celui issu du max. Cela permettra d'avoir un peu de variabilité dans les titres générés tout en gardant la plausibilité des séquences choisies car les caractères les plus probables seront choisis davantage.

[ ]:

1.2 Exercice 2

Il serait intéressant de contrôler l'équilibre variabilité/créativité vs. plausibilité/déterminisme dans la génération de titres. On peut diviser les scores par une température θ avant de faire le softmax pour tasser la distriution. Une valeur de $\theta < 1$ poussera le modèle à prendre moins de risque et générer des caractères plus probables, alors que $\theta > 1$ lui fera prendre plus de risques et générer des séquences moins probables. En général, $\theta = 0.7$ donne des résultats satisfaisants.

Implémentez cette variante et générez 10 séquences avec cette méthode en testant une valeur de θ supérieure à 1 et une valeur inférieure à 1. Qu'observez-vous ?

[ ]:

1.2.1 Exercice 3

Modifier la fonction pour qu'elle prenne en entrée une chaîne de caractères représentant le début d'un titre (par exemple "Star"), et force le réseau à passer par ces caractères (sans oublier `<start>`) avant de générer une fin pour le titre. Ainsi, le modèle fonctionnera comme un "prompt" qui complète les titres de films commencés.

[ ]:

1.3 Pour aller plus loin

- Entraînez ce système sur plus de données issues d'une source différente
- Essayez de varier les hyper-paramètres comme le nombre de couches, échangez les GRU contre des LSTM (attention ces derniers renvoient un état caché sous la forme d'un tuple de deux tenseurs)
- Utilisez l'exemple pytorch `word_language_model` pour entraîner un modèle de langage sur les mots