PSTALN_TP03_textclassif

December 12, 2021

1 Classification de textes : analyse de sentiments

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.

[1]:
```
## example de deploiement local (pas besoin de refaire si vous avez déjà fait␣
 ↪lors du TP1)
# 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
```

L'analyse de sentiment consiste à prédire à partir d'un texte la polarité du sentiment associé. Le système prend en entrée une séquence de mots et génère en sortie une unique prédiction.

Il faut commencer par télécharger les données pour ce problème d'analyse de sentiment. L'archive contient `sanders-twitter-sentiment.csv` qui est un petit corpus de tweets annotés avec une cible (apple, microsoft, google…) et une étiquette de sentiment (positif, negatif, neutre, non pertient).

[2]:
```
%%bash
if [ ! -f sanders-twitter-sentiment.csv ]; then
  wget -q https://pageperso.lis-lab.fr/carlos.ramisch/download_files/pstaln/
 ↪sanders-twitter-sentiment.csv
fi
head -3 sanders-twitter-sentiment.csv
```

```
"1044","125667332931596290","2011-10-16 20:20:01","&quot;3 principal global
players will be active in every market…@Amazon, @Apple, &amp; @Kobo.&quot;
#fbf11 #publaunch @MikeShatzkin http://t.co/1ndxcMO1","neutral","apple"
"71","126384526925639681","2011-10-18 19:49:53","If you've been struggling to
get hold of me, I'm back online with a new iPhone - thanks
@apple","neutral","apple"
"278","126281019476291585","2011-10-18 12:58:35","@azee1v1 @apple @umber Proper
```

```
consolidation, proper syncing, stop losing my PURCHASED items, checkboxes that
do what you think they will do.","negative","apple"
```

Les colonnes qui nous intéressent dans le csv sont les colonnes 4, 5 et 6 contenant le tweet, l'étiquette et la cible.

On peut charger rapidement les données avec la classe python qui lit des csv. Pour faciliter les traitements, nous allons ajouter la cible (`row[5]`) en début de tweet (`row[3]`) entre chevrons. L'étiquette est dans `row[4]`.

Pour vérifier que tout s'est bien passé on peut afficher le nombre d'exemples chargés ainsi qu'un exemple arbitrire.

[3]:
```
texts = []
labels = []

import csv
with open('sanders-twitter-sentiment.csv', 'r', encoding='utf8') as csvfile:
    reader = csv.reader(csvfile, delimiter=',', quotechar='"')
    for row in reader:
        texts.append("<" + row[5] + "> " + row[3])
        labels.append(row[4])

print(len(texts))

print(texts[12])
print(labels[12])
```

```
5513
<apple> Wow I am loving this new @apple update for my touch. #coolness Well done
positive
```

La premier chose à faire est de créer un dictionnaire pour faire correspondre les étiquettes à des entiers. Les étiquettes ainsi converties seront stockées dans la liste python `int_labels`.

[4]:
```
label_vocab = {label: i for i, label in enumerate(set(labels))}
print(label_vocab)

int_labels = [label_vocab[label] for label in labels]
print(int_labels[:10])
```

```
{'positive': 0, 'negative': 1, 'irrelevant': 2, 'neutral': 3}
[3, 3, 1, 2, 3, 0, 2, 3, 3, 2]
```

Nous pouvons opérer de la même manière pour convertir les mots en entiers. Notez que notre système ne fait aucun prétraitement sur le texte du tweet, il se contente de le découper selon les espaces. Un système d'analyse de sentiment plus évolué ferait une tokenisation plus fine, mettrait les mots en minuscues, et irait même jusqu'à les lemmatiser. Notez qu'on utilise un `defaultdict` pour stoquer le vocabulaire. Il attribue un entier à chaque nouveau mot qu'il rencontre. Le mot d'indice 0 est reservé pour un symbole `<eos>` que nous utilserons plus tard.

[5]:
```
import collections
vocab = collections.defaultdict(lambda: len(vocab))
vocab['<eos>'] = 0

int_texts = []
for text in texts:
    int_texts.append([vocab[token] for token in text.split()])

print(int_texts[12])
print(int_labels[12])
```

```
[1, 180, 181, 182, 183, 96, 33, 37, 184, 124, 46, 185, 186, 187, 188]
0
```

On peut vérifier que les mots ont bien été convertis en faisant la conversion dans l'autre sens. Il est interessant de regarder la taille du vocabulaire et la taille maximale d'un tweet dans ce corpus.

[6]:
```
rev_vocab = {y: x for x, y in vocab.items()}
print([rev_vocab[word_id] for word_id in int_texts[12]])

print(len(vocab))
print(max([len(text) for text in int_texts]))
```

```
['<apple>', 'Wow', 'I', 'am', 'loving', 'this', 'new', '@apple', 'update',
'for', 'my', 'touch.', '#coolness', 'Well', 'done']
23541
32
```

Nous sommes prêts à convertir les données en tenseurs pytorch. Il faut importer les modules de pytorch, et définir quelques constantes qui s'assurent que le problème est de taille raisonnable pour tourner sur CPU. La constante `max_len` est importante car les tweets qui ont une taille superieure à cette dernière seront coupés.

[7]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

max_len = 16
batch_size = 64
embed_size = 128
hidden_size = 128
device = torch.device('cpu') # can be changed to cuda if available
```

Afin de rendre les calculs rapides, il est souhaitable de mettre toutes les données dans des tenseurs. Comme les textes sont de taille variable, nous allons les mettre dans des matrices de taille (nombre de tweets, taille maximum d'un tweet). Les tweets trop grands par rapport à la taille maximale fixée seront coupés, et ceux qui sont plus courts seronts garnis de symboles de padding `<eos>` dont la valeur est 0.

Techniquement, il n'est pas nécessaire de mettre toutes les données dans un seul tenseur. En particulier cela devient inefficace quand les textes ont des longueurs très différentes. La seule contrainte est que pour un batch donné, les séquences aient la même taille. De nombreuses bibliothèques (comme `torchtext`) génèrent des batches à la volée qui font la bonne taille.

On notera que `textes[12]` a une longeur de 15 symboles et donc `X[12]` se retrouve paddé avec un symbole `<eos>`.

[8]:
```
X = torch.zeros(len(int_texts), max_len).long()

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

Y = torch.LongTensor(int_labels)

X = X.to(device)
Y = Y.to(device)

print(X.size(), Y.size())
print(X[12], Y[12])
```

```
torch.Size([5513, 16]) torch.Size([5513])
tensor([  1, 180, 181, 182, 183,  96,  33,  37, 184, 124,  46, 185, 186, 187,
        188,   0]) tensor(0)
```

Pour pouvoir estimer les performances du modèle, nous allons diviser les données en un jeu d'entraînement et un jeu de validation.

[9]:
```
X_train = X[:5000]
Y_train = Y[:5000]
X_valid = X[5000:]
Y_valid = Y[5000:]
```

pytorch fournit un générateur de batches qui s'occupe de mélanger les données. Utilisons le pour nos deux sources de données.

[10]:
```
from torch.utils.data import TensorDataset, DataLoader
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)
```

Nous allons d'abord définir une fonction d'évaluation d'un modèle qui calcule le loss moyen sur un ensemble de test, ainsi que le taux d'exemples corrects. Cette fonction utilise l'entropie croisée comme fonction de loss.

Il faut tout d'abord mettre le modèle en mode evaluation pour que les traitements propres à l'apprentissage, comme le dropout, soient désactivés. Il ne faudra pas oublier de le remettre en

mode entrainement lors de l'entraînement.

Puis pour chaque batch produit par le loader, on peut calculer les scores produits par le modèle pour chaque étiquette, en déduire le loss, et calculer les prédictions en prenant l'indice de la classe de score max.

Nous pourrons tester cette fonction lorsque nous aurons créé un modèle.

[11]:
```
def perf(model, loader):
    criterion = nn.CrossEntropyLoss()
    model.eval()
    total_loss = correct = num = 0
    for x, y in loader:
        with torch.no_grad():
            y_scores = model(x)
            loss = criterion(y_scores, y)
            y_pred = torch.max(y_scores, 1)[1]
            correct += torch.sum(y_pred.data == y)
            total_loss += loss.item()
            num += len(y)
    return total_loss / num, correct.item() / num
```

La fonction d'apprentissage n'est pas très différente. Elle contient en plus un optimiseur (Adam est utilisé ici car il a des performances raisonnables lorsqu'on a pas encore exploré les hyper-paramètres). Puis pour chaque époque, on n'oublie pas de remettre le modèle en mode entraînement, et cette fois on parcourt les données d'entraînement batch par batch pendant plusieurs époques.

L'entraînement nécessite de remettre à zero les accumulateurs de gradient dans le modèle, de calculer les scores prédits pour le batch, d'en déduire la fonction de coût, puis de faire la back-propgation. L'optimiseur peut alors appliquer le gradient aux paramètres du modèle.

À la fin de chaque époque, on appelle la fonction `perf` définie précédemment pour se faire une idée des performances sur le jeu de validation.

[12]:
```
def fit(model, epochs):
    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, y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            num += len(y)
        print(epoch, total_loss / num, *perf(model, valid_loader))
```

Le modèle que nous allons créer est un réseau convolutionnel. La convolution permet d'extraire des n-grammes d'embeddings de mots, puis on la suit d'un max-pooling sur la séquence pour que la position des n-grammes soit invariante. La couche Conv1d génère la convolution sur la séquence d'embeddings. Elle attend en entrée un tenseur de taille `(batch_size, embedding_size, sequence_length)`, ce qui va demander de transposer les deux dernières dimensions du tenseur produit par la couche d'embedding. La couche de convolution applique fait un produit entre sa matrice de paramètres et les n-grammes extraits de manière à pouvoir apprendre à sélectionner plusieurs n-grammes. Ses sorties sont de taille `(batch_size, num_filters, sequence_length)`. `kernel_size` permet de régler la taille des n-grammes extraits. La couche de convolution étant linéaire, nous appliquons une fonction non linéaire de type ReLU (rectified linear unit).

La deuxième étape est d'appliquer le max pooling. Il existe une couche de max pooling mais elle est plus adaptée au traitement des images, nous allons donc utiliser la fonction `max_pool1d` qui prend en entrée le tenseur produit par la couche de convolution et la fenêtre sur laquelle appliquer le max (ici, toute la longueur de la séquence). Cette couche renvoie un tenseur de taille `(batch_size, num_filters, num_max_windows)` avec `num_max_windows=1` pour nous. On pourrait imaginer un CNN multi-couches qui fasse progressivement le max sur des sous-fenêtres pour obtenir des représentations plus dépendentes de la position des mots. Il suffit ensuite de redimensionner le tenseur produit par la couche de pooling pour le passer à la couche de décision.

[13]:
```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.embed = nn.Embedding(len(vocab), embed_size)
        self.conv = nn.Conv1d(embed_size, hidden_size, kernel_size=2)
        self.dropout = nn.Dropout(.3)
        self.decision = nn.Linear(hidden_size, len(label_vocab))

    def forward(self, x):
        embed = self.embed(x)
        conv = F.relu(self.conv(embed.transpose(1,2)))
        pool = F.max_pool1d(conv, conv.size(2))
        drop = self.dropout(pool)
        return self.decision(drop.view(x.size(0), -1))

cnn_model = CNN()
cnn_model.to(device)
```

[13]:
```
CNN(
  (embed): Embedding(23541, 128)
  (conv): Conv1d(128, 128, kernel_size=(2,), stride=(1,))
  (dropout): Dropout(p=0.3, inplace=False)
  (decision): Linear(in_features=128, out_features=4, bias=True)
)
```

On peut tester que le modèle renvoie bien une matrice de taille `(batch_size, num_labels)`.

[14]:
```
cnn_model(X[:3])
```

[14]:
```
tensor([[-0.5830, -0.8424, -1.5040, -0.4891],
        [-0.5081, -0.7054, -1.3622, -0.3990],
        [-0.3893, -0.5229, -0.8088, -0.2489]], grad_fn=<AddmmBackward0>)
```

Et l'entraîner avec la fonction `fit`.

[15]:
```
fit(cnn_model, 10)
```

```
0 0.01754871588945389 0.015296203481872179 0.6257309941520468
1 0.013031629955768585 0.013278877457011978 0.6686159844054581
2 0.01099719181060791 0.012060553585242691 0.6978557504873294
3 0.009259150260686874 0.011687587603080661 0.723196881091618
4 0.007772726446390152 0.01141877193117358 0.7290448343079922
5 0.0067715049147605896 0.010980609920353322 0.7504873294346979
6 0.005488927814364433 0.010934719841323201 0.7524366471734892
7 0.004684251172095537 0.010999446709766951 0.7563352826510721
8 0.0038024481773376467 0.011179902133546391 0.7582846003898636
9 0.0032602851539850234 0.011087447097087024 0.7543859649122807
```

1.1 Exercice 1

- Est-ce que le fait de couper les tweets est préjudiciable ? Essayez d'augmenter le paramètre `max_len` pour voir comment le classifieur se comporte.
- Quelle est l'influence de la taille du filtre convolutif et du nombre de filtres appliqués ?
- Jouez sur les tailles des couches, le dropout et la fonction d'activation - qu'observez-vous ?

[ ]:

1.2 Exercice 2

En général, on fait des CNN qui extraient des n-grammes de taille 1 à n (par exemple 1, 2 et 3). Pour celà on met en parallèle plusieurs couches de convolution avec des tailles de filtres différentes, et on concatène les représentations créées après max-pooling. La fonction `torch.cat([x1, x2, ...], dim)` permet de concatèner pluseurs tenseurs sur une dimension donnée.

- Construisez un réseau capable d'extraire des n-grammes de taille 2 et 3 à l'aide de filtres convolutifs de taille multiple. Est-il capable de mieux modéliser la tâche et obtenir de meilleures performances ?

[ ]:

1.3 Exercice 3

Appliquez la même méthodologie que dans ce tutoriel pour le corpus 20newsgroups (http://qwone.com/~jason/20Newsgroups/).

[ ]:

1.4 Pour aller plus loin

- Quel modèle obtient les meilleurs performances ?
- Quelles sont les performances d'un RNN avec 2 couches ? bidirectionnel ?
- Quel est l'impact de la taille du noyeau pour le CNN ? Que se passe-t-il si l'on change la fonction d'activiation après la convolution ?