PSTALN_TP07_bertintent

January 5, 2022

0.1 Embeddings contextuels

- 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.

Notez que pour ce TP nous utilisons les bibliothèques `transformers` que nous n'avons pas utilisées jusqu'à présent.

[1]:
```
## 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
# pip3 install transformers
# python3 -m ipykernel install --user --name=pstaln-env
# jupyter-notebook
## puis sélectionner le noyeau pstaln-evn
```

On s'intéresse ici à la prédiction de l'intention (`intent`) d'un utilisateur dans un système de dialogue. Pour chaque ennoncé, il faut prédire la catégorie de l'ennoncé parmis les 7 intentions spécifiquse au domaine du corpus SNIPS : ajouter un élément à une playlist, réserver un réstaurent, obtenir la météo, écouter de la musique, donner son avis sur un livre, rechercher une oeuvre artistique ou obtenir des horaires de cinéma.

Le corpus est divisé en 3 parties : train, valid et test. Il contient un fichier `seq.in` avec un ennoncé par ligne, `seq.out` avec des étiquettes BIO pour représenter les entités liées aux intentions (non-utilisé ici) et un fichier `label` contenant les intentions.

[4]:
```
%%bash
if [ ! -f snips-nlu/train/label ]; then
    wget -q https://pageperso.lis-lab.fr/carlos.ramisch/download_files/pstaln/
 ↪snips-nlu.zip
fi
unzip -q -o snips-nlu.zip
sort snips-nlu/train/label | uniq -c
```

```
1818 AddToPlaylist
1881 BookRestaurant
1896 GetWeather
1914 PlayMusic
1876 RateBook
1847 SearchCreativeWork
1852 SearchScreeningEvent
```

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

[5]:
```
%matplotlib inline

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import collections
from transformers import DistilBertTokenizer, DistilBertModel
from torch.utils.data import TensorDataset, DataLoader

from matplotlib import pyplot as plt
```

0.1.1 Préparation des données

Commençons par charger les énnoncés et étiquettes correspondant.

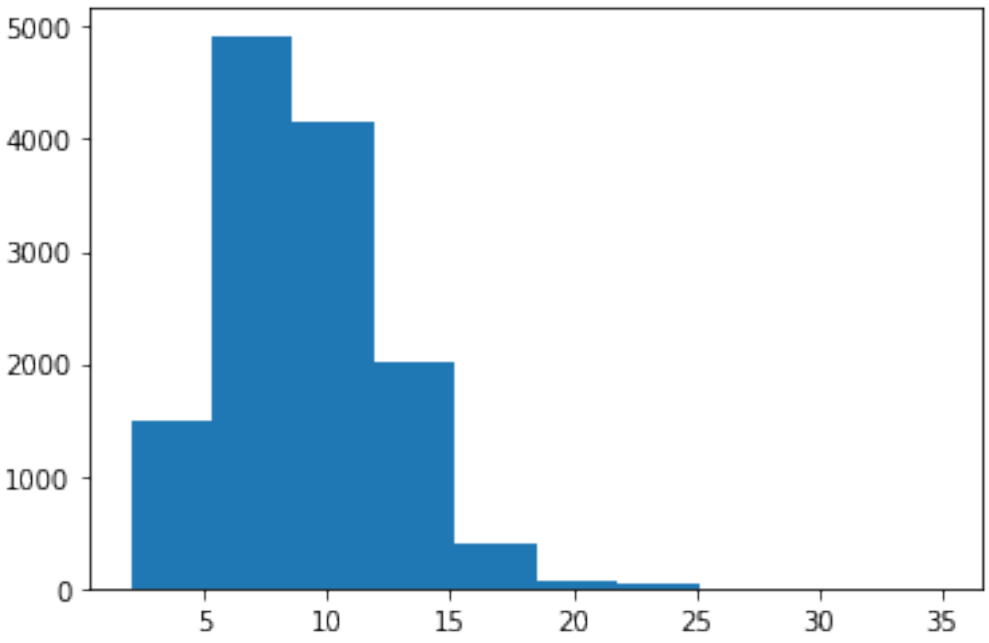
[6]:
```
def load_instances(text_filename, label_filename):
  with open(label_filename) as fp:
    labels = [line.strip() for line in fp]
  with open(text_filename) as fp:
    texts = [line.strip() for line in fp]
  return texts, labels

train_texts, train_labels = load_instances('snips-nlu/train/seq.in', 'snips-nlu/
↪train/label')
valid_texts, valid_labels = load_instances('snips-nlu/valid/seq.in', 'snips-nlu/
↪valid/label')
test_texts, test_labels = load_instances('snips-nlu/test/seq.in', 'snips-nlu/
↪test/label')

for label, text in zip(train_labels[:10], train_texts[:10]):
  print(text, '=>', label)
```

```
listen to westbam alumb allergic on google music => PlayMusic
add step to me to the 50 clásicos playlist => AddToPlaylist
i give this current textbook a rating value of 1 and a best rating of 6 =>
RateBook
```

```
play the song little robin redbreast => PlayMusic
please add iris dement to my playlist this is selena => AddToPlaylist
add slimm cutta calhoun to my this is prince playlist => AddToPlaylist
i want to listen to seventies music => PlayMusic
play a popular chant by brian epstein => PlayMusic
find fish story => SearchScreeningEvent
book a spot for 3 in mt => BookRestaurant
```

Il est intéressant d'afficher un histogramme des longueurs des textes pour voir quelle pourrait être une longueur maximale acceptable pour représenter les données sous forme de tenseurs. Par exemple, une longueur de 20 semble convenir pour la plus grande partie des données.

[7]:
```
plt.hist([len(text.split()) for text in train_texts])
plt.show()
```



Il est temps de définir les classiques paramètres globaux des systèmes. Il faut noter que nous utilisons ici une grande taille de batch. Nous vous conseillons d'utiliser l'accélérateur GPU (à activer sur colab) pour réduire les temps de traitements avec les modèles de type BERT.

[11]:
```
maxlen = 20
batch_size = 256
hidden_size = 128
embed_size = 128
#device = torch.device('cuda')
device = torch.device('cpu')
```

BERT est un modèle issu de la partie encodeur d'un transformer, qui repose essentiellement sur

l'accumulation de couches d'attention avec un encodage de la position. On peut trouver relativement facilement sur le web des explications illustrées sur les détails du fonctionnement de BERT. Nous n'allons pas le réimplémenter, mais utiliser une implémentation créée par la stratup Hugging Face. Le répository de Hugging Face contient une implémentation de diverses variantes de modèles fondés sur les transformers ainsi que des poids préentraînés correspondant.

Le modèle BERT fait une tokenization en sous-mots pour limiter le nombre de mots à traiter, et pour généraliser plus facilement sur les mots inconnus. Nous utiliserons cette tokenization pour toutes les exprériences. Le modèle BERT est très coûteux en temps de calcul et nous utiliserons une version plus petite, appellée DistilBert, issue d'une "distillation" des poids du modèle BERT d'origine.

Il est a noter que le tokenizer divise les mots qu'il ne connait pas en sous-tokens dénotés par **##**, et qu'il faut ajouter un token en début de phrase et en fin de phrase pour la limiter (le modèle est capable de traiter plusieurs phrases).

[9]:
```
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')

def bert_text_to_ids(sentence):
  return torch.tensor(tokenizer.encode(sentence, add_special_tokens=True))

print(tokenizer.tokenize('i really like the band raspigaous.'))
print(bert_text_to_ids('i really like the band raspigaous.'))
```

```
Downloading:   0%|          | 0.00/226k [00:00<?, ?B/s]
Downloading:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
Downloading:   0%|          | 0.00/455k [00:00<?, ?B/s]
Downloading:   0%|          | 0.00/483 [00:00<?, ?B/s]
['i', 'really', 'like', 'the', 'band', 'ras', '##pi', '##ga', '##ous', '.']
tensor([  101,  1045,  2428,  2066,  1996,  2316, 20710,  8197,  3654,  3560,
         1012,   102])
```

Il faut maintenant créer des tenseurs avec les données de chaque sous-ensemble. BERT s'occupe de tokenizer et convertir les mots en identifiants, mais nous devons le faire pour les étiquettes. Nous en profitons pour imposer la longueur maximale et envoyer les tenseurs sur GPU.

[12]:
```
label_vocab = collections.defaultdict(lambda: len(label_vocab))

def prepare_texts(texts, labels):
  X = torch.LongTensor(len(texts), maxlen).fill_(tokenizer.pad_token_id)
  for i, text in enumerate(texts):
    indexed_tokens = bert_text_to_ids(text)
    length = min([maxlen, len(indexed_tokens)])
    X[i,:length] = indexed_tokens[:length]

  Y = torch.tensor([label_vocab[label] for label in labels]).long()
  return X.to(device), Y.to(device)
```

```
X_train, Y_train = prepare_texts(train_texts, train_labels)
X_valid, Y_valid = prepare_texts(valid_texts, valid_labels)
X_test, Y_test = prepare_texts(test_texts, test_labels)

print(X_train.shape)
print(X_train[:3])
```

```
torch.Size([13084, 20])
tensor([[  101,  4952,  2000,  2225,  3676,  2213,  2632, 25438, 27395,  2006,
          8224,  2189,   102,     0,     0,     0,     0,     0,     0,     0],
        [  101,  5587,  3357,  2000,  2033,  2000,  1996,  2753, 18856, 21369,
         13186,  2377,  9863,   102,     0,     0,     0,     0,     0,     0],
        [  101,  1045,  2507,  2023,  2783, 16432,  1037,  5790,  3643,  1997,
          1015,  1998,  1037,  2190,  5790,  1997,  1020,   102,     0,     0]])
```

Il s'agit maintenant de créer des datasets et générateurs de batches en utilisant l'approche classique en pytorch.

[13]:
```
train_set = TensorDataset(X_train, Y_train)
valid_set = TensorDataset(X_valid, Y_valid)
test_set = TensorDataset(X_test, Y_test)

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

0.1.2 Modèle RNN

Le premier modèle est un RNN bidirectionnel à base de GRU très classique. Il commence par une couche d'embeddings initialisés aléatoirement, suivie d'une couche de GRU, suivi de dropout et d'une couche de décision. Nous utilisons le dernier état caché du RNN comme représentation donnée à la couche de décision, dont la forme de tenseur est (nombre de couches GRU, nombre de directions, taille de l'état caché). On en déduit une taille de couche cachée de $1 \times 2 \times$ hidden_size.

La fonction `forward` est très classique et nécessite juste de manipuler le dernier état caché du RNN pour qu'il ait comme forme (taille du batch, taille de l'état caché $\times$ directions $\times$ nombre de couches). Cette fonction prend en entrée un tenseur x représentant un batch de taille (taille du batch, longueur maximale d'une phrase).

Il est toujours bien de vérifier que le tenseur renvoyé par l'inférence a pour forme (taille du batch, nombre de classes).

[14]:
```
class RNNClassifier(nn.Module):
  def __init__(self):
    super().__init__()
    self.embed = nn.Embedding(tokenizer.vocab_size, embed_size,␣
 ↪padding_idx=tokenizer.pad_token_id)
```

```
    self.rnn = nn.GRU(embed_size, hidden_size, num_layers=1,␣
↪bidirectional=True, batch_first=True)
    self.dropout = nn.Dropout(0.3)
    self.decision = nn.Linear(1 * 2 * hidden_size, len(label_vocab))
    self.to(device)

  def forward(self, x):
    embed = self.embed(x)
    output, hidden = self.rnn(embed)
    drop = self.dropout(hidden.transpose(0, 1).reshape(x.shape[0], -1))
    return self.decision(drop)

rnn_model = RNNClassifier()
with torch.no_grad():
  print(rnn_model(X_train[:3]).shape)
```

```
torch.Size([3, 7])
```

La fonction de calcul des performances est classique. Elle utilise un critère d'entropie croisée (même loss qu'en entrînement), et calcule le taux de corrects du modèle.

[15]:
```
def perf(model, loader):
    criterion = nn.CrossEntropyLoss()
    model.eval()
    total_loss = num = correct = 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 == y).item()
        total_loss += loss.item()
        num += len(y)
    return total_loss / num, correct / num

perf(rnn_model, valid_loader)
```

[15]:
```
(0.00839418717793056, 0.13)
```

La fonction d'apprentissage est elle aussi standard. Nous rendons toutefois le taux d'apprentissage (lr) paramètrable car le fine-tuning des paramètres de BERT avec Adam est instable si le taux d'apprentissage n'est pas plus bas que la valeur par défaut. Cette dernière marche très bien pour le RNN.

[16]:
```
def fit(model, epochs, lr=1e-3):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    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))

fit(rnn_model, 10)
```

```
0 0.0035654590301413816 0.0010006012128932135 0.9442857142857143
1 0.0005780119108373839 0.0005114384101969856 0.9671428571428572
2 0.00029595708645349504 0.00043314388820103235 0.9728571428571429
3 0.00017616987700745787 0.0004401799078498568 0.9685714285714285
4 0.00012116501221985071 0.00041220438799687795 0.9742857142857143
5 7.75945428400111e-05 0.00047692500054836274 0.97
6 5.41476529329054e-05 0.00044398205620901926 0.9757142857142858
7 4.0845478067833116e-05 0.0005884518900087901 0.9714285714285714
8 2.4365925317851114e-05 0.0004921638646296092 0.9728571428571429
9 1.1856131834606946e-05 0.0004888819264514105 0.9771428571428571
```

0.1.3 Modèle BERT

Le second modèle est basé sur DistilBERT. L'idée est de faire passer les entrées (tokens) à travers l'encodeur d'un transformer (BERT), d'en récupérer la représentation aggrégée sur toute la phrase (équivalent de l'état caché du RNN après le dernier mot) et de la passer à une couche de décision. Il faut faire attention à dimensionner cette dernière avec la taille de la représentation générée par BERT.

Dans la fonction forward, BERT nécessite un masque pour désigner les mots sur lesquels porte le mécanisme d'attention, sans quoi les représentations générées n'auront pas de sens. Le composant renvoie une représentation par mot et nous utilisons le max pooling pour aggréger ces représentations.

[17]:
```
class BertClassifier(nn.Module):
  def __init__(self):
    super().__init__()
    self.bert = DistilBertModel.from_pretrained('distilbert-base-uncased')
    self.decision = nn.Linear(self.bert.config.hidden_size, len(label_vocab))
    self.to(device)

  def forward(self, x):
    output = self.bert(x, attention_mask = (x != tokenizer.pad_token_id).long())
    return self.decision(torch.max(output[0], 1)[0])
```

```
bert_model = BertClassifier()
with torch.no_grad():
  print(bert_model(X_train[:3]).shape)
```

```
Downloading:   0%|          | 0.00/256M [00:00<?, ?B/s]
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_transform.weight',
'vocab_transform.bias', 'vocab_layer_norm.bias', 'vocab_projector.weight',
'vocab_projector.bias', 'vocab_layer_norm.weight']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
torch.Size([3, 7])
```

Étant donné la longueur de l'entraînement avec BERT, nous ne ferons que 5 époques avec un taux d'apprentissage de 0.00002.

[ ]:
```
fit(bert_model, 5, lr=2e-5)
```

Les résultats obtenus peuvent être comparés à ceux de https://github.com/czhang99/Capsule-NLU. Normalement ils devraient pas être très loin de l'état de l'art sur ce corpus.

[ ]:
```
print('RNN', *perf(rnn_model, test_loader))
print('BERT', *perf(bert_model, test_loader))
```

0.1.4 Exercice 1

Que se passe-t-il si l'on n'autorise pas la partie BERT du modèle à modifier ses poids (mettre `requires_grad=False` à tous les paramètres du module récupérable avec `module.parameters()`) ? Testez et expliquez.

[ ]:
```

```

0.1.5 Exercice 2

Pour combiner les embeddings des mots générés pour chaque position de l'entrée, nous avons fait un max-pooling.
- Quelle est la meilleure méthode de pooling parmi (min, max, mean) ? Testez et comparez - Au lieu de faire du pooling, on peut implémenter un RNN pour aggréger les représentations de BERT sur les mots. Testez et comparez.

[ ]:
```

```

0.1.6 Pour aller plus loin

- Faire un système BERT pour les prédictions d'entités BIO présentes dans les fichiers `seq.out`