PSTALN_TP05_tagging

December 15, 2021

1 Étiquetage de séquences et segmentation

- 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 la bibliothèque `pytorch_crf` que nous n'avons pas utilisée 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 pytorch_crf
# python3 -m ipykernel install --user --name=pstaln-env
# jupyter-notebook
## puis sélectionner le noyeau pstaln-evn
```

Nous allons entraîner un étiqueteur de séquences à prédire les groupes nominaux représentant des entités visuelles dans une phrase. Les données sont extraites du corpus Flicker30k entities et le problème est formalisé comme la prédiction d'une étiquette par mot indiquant si une entité commence sur le mot (B), si elle continue (I) ou si on est à l'exterieur d'une entité. Le type de modèle que nous allons concevoir pourrait être appliqué à d'autres problèmes d'étiquetage, p.ex. les taggeurs en parties de discours.

Nous allons entraîner un RNN à faire ces prédictions et les embeddings de mots seront initialisés avec des embeddings préentrainés avec fasttext. Nous en profiterons pour voir comment on peut créer un RNN moins dépendant de la taille des séquences et capable de traiter en test des séquences plus longues que celles qu'il a vu en entraînement.

Les données sont sous la forme d'un mot par ligne suivi de son étiquette. Les phrases sont séparées par des lignes vides. C'est un format tabulaire assez classique en TAL (p.ex. le format CoNLL-U de UD).

[2]:
```
%%bash
# Corpus annoté en entités
if [ ! -f f30kE-captions-bio.txt ]; then
    wget -q https://pageperso.lis-lab.fr/carlos.ramisch/download_files/pstaln/
 ↪f30kE-captions-bio.txt
fi
# Embeddings pré-entraînés (cf. plus bas)
if [ ! -f wiki.en.filtered.vec ]; then
  wget -q https://pageperso.lis-lab.fr/carlos.ramisch/download_files/pstaln/
 ↪wiki.en.filtered.vec.gz -O - | gunzip > wiki.en.filtered.vec
fi
head -16 f30kE-captions-bio.txt
```

```
A B
group I
of I
5 I
scuba I
divers I
talk O
on O
the B
surface I
next O
to O
a B
barrier I
island I
. O
```

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

[3]:
```
import collections
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
from torchcrf import CRF ## NOUVEAU

device = torch.device('cpu') # can be changed to cuda if available
```

1.0.1 Chargement des données

La première chose à faire est charger les données. Un premier tableau contiendra les mots de chaque phrase, et un second les étiquettes associées.

[4]:
```
texts = []
labels = []

with open('f30kE-captions-bio.txt', encoding="utf-8") as fp:
    text = []
    label = []
    for line in fp:
        tokens = line.strip().split()
        if len(tokens) == 0:
            texts.append(text)
            labels.append(label)
            text = []
            label = []
        else:
            text.append(tokens[0])
            label.append(tokens[1])

print(len(texts))

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

```
5500
['A', 'man', 'leans', 'against', 'a', 'pay', 'phone', 'while', 'reading', 'a',
'paper', '.']
['B', 'I', 'O', 'O', 'B', 'I', 'I', 'O', 'O', 'B', 'I', 'O']
```

Il faut ensuite convertir les étiquettes et les mots en entiers. Nous allons devoir créer des séquences de taille fixe, donc il faut réserver une étiquette de padding, ici `<eos>`. Toutefois, pour l'instant nous nous contentons de convertir les étiquettes sans appliquer le padding. Un `defaultdict` est utilisé pour créer le vocabulaire des étiquettes. À chaque fois qu'une nouvelle étiquette est rencontrée, il l'associe à une nouvelle valeur.

[5]:
```
# Astuce : defaultdict renvoie un nouvel entier à chaque accès d'une valeur␣
↪absente
label_vocab = collections.defaultdict(lambda: len(label_vocab))
label_vocab['<eos>'] = 0

int_labels = []
for label in labels:
    int_labels.append([label_vocab[token] for token in label])

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

```
[1, 2, 3, 3, 1, 2, 2, 3, 3, 1, 2, 3]
defaultdict(<function <lambda> at 0x7fc781b284c0>, {'<eos>': 0, 'B': 1, 'I': 2,
'O': 3})
```

Il en va de même pour les textes à qui nous allons dédier un vocabulaire différent de celui des étiquettes. Les embeddings préentraînés seront chargés à partir de ce vocabulaire. Il faut veiller à mettre les mots des textes en minuscules car le vocabulaire des embeddings fasttext est en minuscules.

[6]:
```
vocab = collections.defaultdict(lambda: len(vocab))
vocab['<eos>'] = 0

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

print(int_texts[12])
len(vocab)
```

```
[1, 16, 85, 86, 1, 87, 88, 36, 89, 1, 90, 15]
```

[6]:
```
4596
```

Nous pouvons maintenant créer des vocabulaires inversés pour pouvoir revenir vers les étiquettes et mots. Celà et utile pour vérifier le contenu d'un tenseur, et surtout pour convertir les décisions du système en étiquettes textuelles à la fin, une fois que le système sort des prédictions.

[7]:
```
rev_label_vocab = {y: x for x, y in label_vocab.items()}
rev_vocab = {y: x for x, y in vocab.items()}
```

1.0.2 Pré-traitement des données

Les constantes suivantes définissent des hyperparamètres de notre modèle : - `max_len` est la longueur maximale d'une phrase en apprentissage - `batch_size` est la taille des batches - `embed_size` est la taille des embeddings préentraînés (nous utiliserons les embeddings téléchargeables sur le site de fasttext qui sont de taille 300. - `hidden_size` est la taille de l'état caché du RNN

[8]:
```
max_len = 16
batch_size = 64
embed_size = 300 # modifier paramètre uniquement si vous utilisez d'autres␣
↪embeddings pré-entraînés
hidden_size = 128
```

Il faut maintenant créer des tenseurs pytorch avec les données phrases et les étiquettes associées. X et Y sont des tenseurs d'entiers de taille (le nombre d'exemples du corpus, la longueur maximale d'une phrase). Ils sont initialisés à zéro car c'est la valeur de padding.

Pour chaque phrase et séquence d'étiquettes associées, nous calculons la longueur effective à intégrer en fonction de la longueur maximale (les phrases et séquences d'étiquettes trop longues sont coupées). Puis nous pouvons les intégrer dans les tenseurs en début de ligne, les fins de lignes étant remplies de zéros (`<eos>`).

[9]:
```
X = torch.zeros(len(int_texts), max_len).long()
Y = torch.zeros(len(int_labels), max_len).long()

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

print(X[12])
print(Y[12])
```

```
tensor([ 1, 16, 85, 86,  1, 87, 88, 36, 89,  1, 90, 15,  0,  0,  0,  0])
tensor([1, 2, 3, 3, 1, 2, 2, 3, 3, 1, 2, 3, 0, 0, 0, 0])
```

Pour vérifier les performances du système, nous le séparons en un ensemble d'entraînement et un ensemble de développement. Comme il y a 5500 exemple, nous utiliserons les 5000 premiers pour l'entraînement et les 500 suivants pour la validation.

[10]:
```
X_train = X[:5000]
Y_train = Y[:5000]
X_valid = X[5000:] # il reste 500 examples pour la validation
Y_valid = Y[5000:]
```

Pytorch contient des classes permettant facilement de charger des batches d'exemples déjà mélangés pour l'entraînement. Nous allons en tirer parti, comme d'habitude.

[11]:
```
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)
```

1.0.3 Embeddings pré-entraînés

La prochaine étape consiste en le chargement des embeddings préentraînés. Les embeddings fasttext peuvent être téléchargés depuis la page suivante : https://fasttext.cc/docs/en/english-vectors.html

Toutefois, ces embeddings couvrant plusieurs millions de mots, le fichier fait plusieurs GiB et une version filtrée par rapport au vocabulaire du corpus est disponible. Notez qu'en production, il faudrait utiliser tous les embeddings pour avoir une bonne couverture du vocabulaire que l'on peut rencontrer.

Les embeddings ont le format suivant : - chaque ligne décrit l'embedding d'un mot - le mot est sur la première colone, suivi des valeurs du vecteur dans l'espace de 300 dimension.

Nous créons donc un tenseur à zéro, puis plaçons l'embedding de chaque mot du vocabulaire rencontré à la bonne ligne dans le tenseur. Les embeddings des mots non couverts dans ce fichier resteront à zéro (question : combien sont-ils ?)

[12]:
```
pretrained_weights = torch.zeros(len(vocab), 300)
with open('wiki.en.filtered.vec', encoding="utf-8") as fp:
    for line in fp:
        tokens = line.strip().split()
        if tokens[0] in vocab:
            pretrained_weights[vocab[tokens[0]]] = torch.FloatTensor([float(x)␣
↪for x in tokens[1:]])

pretrained_weights[12][:10]
```

[12]:
```
tensor([-0.0159,  0.2218, -0.0831,  0.3253, -0.0533,  0.1563,  0.3345,  0.0572,
        -0.1407, -0.0389])
```

1.0.4 Étiqueteur

Le modèle est une class qui étend `nn.Module`. Son constructeur appelle le constructeur de la class mère, puis déclare les différentes couches : une couche d'embeddings, une couche de GRU et une couche de décision. Cette couche de décision sera appliquée à chaque position de la phrase pour prédire l'étiquette associée.

Le premier détail important que nous allons traiter est le problème du padding. Pour pouvoir prendre en entrée des batches de phrases de la même taille, nous avons complété ces dernières avec des 0. Le mot d'indice zéro a un embedding et le système peut donc l'utiliser pour apprendre des régularités des données. Si on utilise un RNN bidirectionnel, l'état caché après chaque padding peut aussi contribuer à la représentation créée. Ces deux problème font que le modèle aura un comportement différent si on change la taille des séquences traitées.

Une solution est de forcer l'état caché à rester à zéro en présence de padding. Pour celà il faut spécifier le `padding_idx` de la couche d'emebdding pour la représentation associée au padding soit toujours le vecteur nul. Ensuite, dans le RNN, l'état caché est calculé comme une transformation affine à partir de l'embedding en entrée et de l'état caché précédent. Comme l'état caché initial est à zéro, et que l'embedding du padding est à zéro, si on désactive le bias dans les calculs (en donnant l'option `bias=False` au GRU), cela va forcer l'état caché à rester à zéro tout au long du padding. Le modèle peut ainsi traiter des séquences de taille arbitraire, même quand il est bidirectionnel.

Le deuxième détail est l'initialisation des embeddings. La méthode diffère selon les versions de pytorch, mais la façon présentée ici marche dans la plupart des cas. Nous allons directement modifier le champ `weight` de la couche d'embeddings et remplacer ce paramètre par nos embeddings préentraînés. `requires_grad=False` permet de geler la couche d'embeddings et donc de ne pas les modifier lors de l'apprentissage. Cela permettra en prédiction d'utiliser des embeddings fasttext pour les mots que nous n'avons pas observés dans le corpus d'apprentissage, en espérant qu'un mot d'embedding proche s'y trouvait. Si on omet cette option, les embeddings sont fine-tunés pour maximiser les performances du système.

[13]:
```
class RNN(nn.Module):
    def __init__(self, pretrained_weights, label_vocab):
        super().__init__()
        self.embed = nn.Embedding(len(vocab), embed_size,␣
↪padding_idx=vocab['<eos>'])
```

```
        self.embed.weight = nn.Parameter(pretrained_weights,␣
↪requires_grad=False)
        self.rnn = nn.GRU(embed_size, hidden_size, bias=False, num_layers=1,␣
↪bidirectional=False, batch_first=True)
        self.dropout = nn.Dropout(0.3)
        self.decision = nn.Linear(hidden_size * 1 * 1, len(label_vocab))

    def forward(self, x):
        embed = self.embed(x)
        output, hidden = self.rnn(embed)
        return self.decision(self.dropout(output))

rnn_model = RNN(pretrained_weights, label_vocab)
rnn_model
```

[13]:
```
RNN(
  (embed): Embedding(4596, 300, padding_idx=0)
  (rnn): GRU(300, 128, bias=False, batch_first=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (decision): Linear(in_features=128, out_features=4, bias=True)
)
```

On peut vérifier que le modèle génère des prédictions comme attendu en passant un petit batch de 2 exemples comme entrée (le contexte `no_grad()` desactive le calcul du gradient, pour économiser des ressources quand le modèle n'est pas en mode entraînement).

Nous pouvons remarquer que la taille du tenseur produit est `(batch_size, sequence_length, num_labels)`, c'est-à-dire, le résultat est de dimension 2 (taille du batch) par 16 (longueur de la séquence) par 4 (nombre de classes à prédire), prédisant bien une distribution sur les classes par mot en entrée.

[14]:
```
with torch.no_grad():
  print(rnn_model(X[:2]).shape)
```

```
torch.Size([2, 16, 4])
```

1.0.5 Entraînement du modèle

La fonction qui calcule les performances d'un modèle doit accomoder la nouvelle forme des données. En particulier, le critère d'entropie croisée n'accepte que des matrices à deux dimensions, donc il faut redimentionner les scores produits pour qu'ils ait la taille `(batch_size * sequence_length, num_labels)` et les références pour quelles aient la taille `(batch_size * sequence_length)`.

Ensuite, il faut modifier le max qui calcule les prédictions pour qu'il agisse sur la dernière dimension du tenseur `y_scores`.

Et finalement, pour calculer le score d'une séquence, nous devons ignorer le padding. Pour celà une matrice `mask` est crée qui contient 1 pour les éléments non nuls de la matrice contenant les étiquettes et 0 sinon. On peut calculer le nombre de corrects ainsi que le numérateur de la fonction de performance en appliquant le masque.

La loss, pour être comparable au loss de l'entraînement, n'est pas modifié.

[15]:
```
def perf(model, loader):
    criterion = nn.CrossEntropyLoss()
    model.eval()
    total_loss = correct = num_loss = num_perf = 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)))
        y_pred = torch.max(y_scores, 2)[1]
        mask = (y != 0)
        correct += torch.sum((y_pred.data == y) * mask)
        total_loss += loss.item()
        num_loss += len(y)
        num_perf += torch.sum(mask).item()
    return total_loss / num_loss, correct.item() / num_perf

perf(rnn_model, valid_loader)
```

[15]:
```
(0.022371821880340575, 0.2463002114164905)
```

La fonction d'apprentissage est modifiée en deux endroits. Tout d'abord, pytorch refuse d'entraîner un modèle contenant des paramètres sans gradient. Nous devons donc filtrer la liste des paramètres passés à l'optimiseur pour qu'elle ne contienne pas les embeddings "gelés".

Ensuite, nous appliquons le même redimenssionnement des scores et des étiquettes pour accomoder le critère d'apprentissage.

[16]:
```
def fit(model, epochs, train_loader, valid_loader):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(filter(lambda param: param.requires_grad, 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(rnn_model, 10, train_loader, valid_loader)
```

```
0 0.008164683812856675 0.002341021880507469 0.9291754756871036
1 0.0015561414018273353 0.0013825022578239441 0.9633544749823819
2 0.001048680441081524 0.0011660869047045708 0.9695207892882312
3 0.0008526438679546118 0.0010801054909825324 0.971106412966878
4 0.0007580528136342764 0.0010261845737695695 0.9733967582804792
5 0.0006757109813392162 0.000987800732254982 0.9735729386892178
6 0.0006470875736325979 0.0010101670026779175 0.9726920366455251
7 0.0005802143888548017 0.0009862197488546371 0.9725158562367865
8 0.0005356406919658184 0.0010093320086598397 0.9732205778717407
9 0.0005201084356755018 0.0009748029783368111 0.9753347427766033
```

1.0.6 Modèle CRF

Une des limitations du modèle ci-dessus est le choix d'une étiquette par mot en maximisant la probabilité de cette étiquette de manière indépendante des autres étiquettes. Or, nous savons que dans cette tâche, il y a des interdépendances entre les étiquettes. Par exemple, on ne peut pas prédire un `I` sans qu'aucun `B` n'ait été prédit. Ces contraintes sur les séquences d'étiquettes possibles ne sont pas prises en compte, ni lors de l'apprentissage, ni lors de la prédiction sur le jeu de validation.

Pour palier cette limitation, on souhaite mainenant ajouter à ce modèle une couche de décision de type Conditional Random Field (CRF). Un CRF est un modèle probabiliste d'un étiquetage y d'une séquence x définit par :

$$P(y|x) = \frac{1}{Z(x)} \prod_i \exp\left(\psi_i(y, x)\right) \quad (1)$$

$$Z(x) = \sum_{y'} \prod_i \exp\left(\psi_i(y', x)\right) \quad (2)$$

où ψ_i donne un score au voisinage du slot i en fonction d'une clique de y et de x (potentiellement dans son intégralité), et Z est un facteur de normalisation pour obtenir une probabilité.

Pour une séquence linéaire, une façon populaire de construire le voisinage est de décomposer ψ_i en deux fonctions : f donne un score à la transition (y_i, y_{i-1}), g donne un score à l'émission (y_i, x).

$$\psi_i = f(y_i, y_{i-1}) + g(y_i, x) \quad (3)$$

Dans notre modèle, f sera un jeu de paramètres avec un paramètre pour chaque couple d'étiquettes, et g utilisera les scores fournis par un RNN tel que celui utilisé précédemment. Les paramètres de f et g sont appris sur les données, c'est-à-dire, la fonction d'émission g sera apprise par les paramètres du RNN, alors que les scores de transition f sont des paramètres du CRF, appris sur les séquences d'étiquettes observées dans les données d'entraînement aussi.

Classiquement (et bien avant les modèles neuronaux en TAL), on entraînait un CRF en maximisant

la log-vraisemblance des données :

$$LL = \sum_{(x,y)} \log P(y|x) \tag{4}$$

$$= \sum_{(x,y)} -\log Z(x) + \sum_i \psi_i(y,x) \tag{5}$$

Dans le cadre de modèles neuronaux avec pytorch, cela revient à minimiser $-LL$ (car un modèle neuronal est entraîné à *minimiser* une fonction de perte, et non pas à *maximiser* une (log-)vraisemblance).

Nous utiliserons ici le module pytorch-crf (documentation à https://pytorch-crf.readthedocs.io). Notez que nos tenseurs ont pour première dimension le batch, ce qu'il faut spécifier à la classe `CRF`.

[17]:
```
crf_model = CRF(len(label_vocab), batch_first=True)
```

On peut tester le modèle. Ce dernier contient deux fonctions : * la fonction `forward()` calcule la log likelihood LL en fonction des fonctions f et g donnant des scores de transition et d'émission * la fonction `decode()` retourne la séquence d'étiquettes de plus haute probabilité avec l'algorithme de programmation dynamique Viterbi

Ces deux fonctions peuvent prendre un masque en paramètre pour indiquer le padding (ici l'étiquette 0). Elles prennent en entrée un tenseur représentant les emissions calculées à partir des entrées. Nous les produirons avec le modèle RNN inchangé. Cependant, pour que l'apprentissage de fasse de bout en bout en prenant en compte le CRF, nous allons réinitialiser notre modèle RNN (cf. plus bas) au lieu d'utiliser celui déjà pré-entraîné pour choisir une étiquette par mot indépendemment des autres.

[18]:
```
with torch.no_grad():
  mask = (Y[:2] != 0)
  print('log likelihood', crf_model(rnn_model(X[:2]), Y[:2], mask=mask))
  pred = crf_model.decode(rnn_model(X[:2]), mask=mask)
  print('viterbi output', pred)
  print('BIO labels of 1st example', [rev_label_vocab[l] for l in pred[0]] )
```

```
log likelihood tensor(-0.6047)
viterbi output [[1, 2, 2, 2, 2, 2, 3, 3, 1, 2, 3, 3, 1, 2, 2, 3], [1, 3, 1, 2,
3, 1, 2, 2, 2, 3, 1, 2, 3]]
BIO labels of 1st example ['B', 'I', 'I', 'I', 'I', 'I', 'O', 'O', 'B', 'I',
'O', 'O', 'B', 'I', 'I', 'O']
```

1.0.7 Entraînement du CRF

Le calcul des performances sur le jeu de validation est similaire à celui pour le RNN, à la différence qu'il faut utiliser `decode()` pour obtenir la séquence d'étiquettes prédites et calculer le nombre d'étiquettes correctes manuellement.

Concernant la loss, elle est calculée par le modèle CRF comme étant la log-vraisemblance moyenne par phrase de la probabilité conditionnelle de la séquence d'étiquettes étant donné l'entrée. Comme on cherche à minimiser la loss, il faut cumuler la log-vraisemblance *négative* renvoyée.

[19]:
```
def perf_crf(emission_model, crf_model, loader):
    def num_correct(y_pred, y):
      correct = 0
      for y_pred_current, y_current in zip(y_pred, y):
          for i in range(len(y_pred_current)):
            if y_pred_current[i] == y_current[i]:
              correct += 1
      return correct

    emission_model.eval()
    crf_model.eval()
    loss = num_loss = correct = num_perf = 0
    for x, y in loader:
      with torch.no_grad():
        mask = (y != 0)
        emissions = emission_model(x)

        y_pred = crf_model.decode(emissions, mask)
        correct += num_correct(y_pred, y)
        num_perf += torch.sum(mask).item()

        loss -= crf_model(emissions, y, mask).item()
        num_loss += len(y)
    return loss / num_loss, correct / num_perf

perf_crf(rnn_model, crf_model, valid_loader)
```

[19]:
```
(0.9597019195556641, 0.9749823819591261)
```

L'apprentissage du modèle contenant le CRF doit aussi être modifié pour tenir compte du fait que le loss est spécifique à ce module. Notez que l'on doit appliquer l'optimiseur sur les paramètres des deux modèles, et que `crf_model.forward()`, appelée implicitement par `crf_model()`, renvoie la log-vraisemblence positive alors que l'on souhaite minimiser la log-vraisemblance negative.

Une meilleure façon d'implémenter le CRF serait de construire une classe incluant les deux modèles (émissions et CRF) de manière à écrire les fonctions `perf()` et `fit()` de manière plus naturelle.

[20]:
```
def fit_crf(emission_model, crf_model, epochs, train_loader, valid_loader):
    all_parameters = list(emission_model.parameters()) + list(crf_model.
 ↪parameters())
    optimizer = optim.Adam(filter(lambda param: param.requires_grad,␣
 ↪all_parameters))
    for epoch in range(epochs):
        emission_model.train()
        crf_model.train()
        total_loss = num = 0
        for x, y in train_loader:
            optimizer.zero_grad()
```

```
            mask = (y != 0)
            emissions = emission_model(x)
            loss = -crf_model(emissions, y, mask)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            num += len(y)
        print(epoch, total_loss / num, *perf_crf(emission_model, crf_model,␣
 ↪valid_loader))

rnn_model = RNN(pretrained_weights, label_vocab)
crf_model = CRF(len(label_vocab), batch_first=True)
fit_crf(rnn_model, crf_model, 10, train_loader, valid_loader)
```

```
0 5.692288826370239 1.847680908203125 0.9445031712473573
1 1.2708396858215332 1.2025685958862304 0.963706835799859
2 0.9021796020507813 1.0808255920410157 0.9677589852008457
3 0.7445825031280517 0.9716341171264649 0.9721634954193094
4 0.6685708683013916 0.9184101486206054 0.9732205778717407
5 0.5960258193969726 0.8864087142944336 0.9737491190979564
6 0.5489894599914551 0.9193168106079102 0.9733967582804792
7 0.5005862182617188 0.8888548965454102 0.9737491190979564
8 0.4678652111053467 0.8576353683471679 0.9758632840028189
9 0.43000480155944826 0.8797459335327148 0.9760394644115574
```

À la fin de l'entraînement, il est possible de visualiser les paramètres appris par le modèle pour la fonction f donnant des scores aux transitions. On voit que les transitions de plus haut score sont $B \to I$, $I \to I$ et $O \to B$. De la même manière, $O \to I$ est la plus pénalisée.

[21]:
```
print(label_vocab)
print(crf_model.transitions)

# create a list with (label1, label2, transition-score)
values = [(rev_label_vocab[i], rev_label_vocab[j], crf_model.transitions[i, j].
 ↪item()) for i in range(4) for j in range(4)]

# sort and show highest values
values.sort(key=lambda x: -x[2])
print("* Plus probables :")
for i, j, value in values[:3]:
  print(i + '->' + j, value)
print("* Moins probables :")
for i, j, value in values[-3:]:
  print(i + '->' + j, value)
```

```
defaultdict(<function <lambda> at 0x7fc781b284c0>, {'<eos>': 0, 'B': 1, 'I': 2,
'O': 3})
Parameter containing:
```

```
tensor([[-0.0775, -0.1294,  0.0328,  0.0273],
        [-0.0133, -0.2858,  0.0603,  0.0071],
        [-0.1086, -0.1328,  0.1566,  0.0626],
        [-0.0629,  0.1881, -0.1628,  0.0381]], requires_grad=True)
* Plus probables :
O->B 0.18812265992164612
I->I 0.1566227376461029
I->O 0.0625733807682991
* Moins probables :
I->B -0.13281172513961792
O->I -0.16284632682800293
B->B -0.28584223985671997
```

Une fois le modèle entraîné, nous pouvons écrire une fonction qui génère les prédictions pour une phrase quelconque.

1.1 Exercice 1

Vous pouvez essayer d'améliorer le modèle en utilisant une couche GRU bidirectionnelle à la place de la couche unidirectionnelle. Attention, la sortie de la couche GRU sera alors deux fois plus grande, car elle contient la concaténation des GRU dans les deux directions.

Une autre architecture populaire est le *stacked LSTM* ou, dans notre cas, *stacked GRU*. Testez le modèle avec une architecture à deux couches.

Est-ce que ces variantes apprennent plus vite ? Est-ce qu'elles obtiennent de meilleures performances sur le jeu de validation ?

[ ]:

1.2 Exercice 2

1. Écrire une fonction `predict(sentence)` qui prend en entrée une phrase sous la forme d'une chaîne de caractères et renvoie une liste de couples (étiquette, mot). Cette fonction devra :

- convertir les mots en entiers (nous n'avons pas prévu de symbole pour les mots inconnus donc ils seront remplacés par du padding)
- faire un batch de taille 1 sous la forme d'un tenseur de dimensions (1, longueur de la phrase)
- calculer les scores des étiquettes à chaque position
- calculer les prédictions correspondantes
- et convertir les entiers prédits en text grâce au dictionnaire inversé créé plus haut

Notez que cette fonction peut prendre en entrée des phrases plus longues que `max_len` et que le temps de calcul n'est pas gaspillé par le padding.

[ ]:

1.3 Exercice 3

Créer un système de reconnaissance d'entités nommées à partir des données de CoNLL-2003 (https://github.com/davidsbatista/NER-datasets/tree/master/CONLL2003). Vous utiliserez

uniquement les formes (1e colone) et créerez une fonction prenant une chaîne de caractères en entrée et renvoyant la séquence d'entités nommées trouvées. Vous pouvez vous aider de Spacy pour la tokenization (https://spacy.io/api/tokenizer).

[ ]:

1.4 Exercice 4

Le TP sur l'analyse de sentiments (classification de textes) utilisait un modèle convolutif qui ne prend pas en compte l'ordre des mots. Créez un modèle alternatif qui, au lieu d'extraire des n-grammes pertinents à l'aide d'un CNN, représente chaque tweet avec un réseau récurrent GRU (ou LSTM, au choix). Vous pouvez aussi essayer de combiner les deux, avoir une représentation par GRU et une représentation par CNN, concaténer les deux avant la couche dense de décision.

[ ]:

1.5 Exercice 5

Utilisez une stratégie de gestion des mots inconnus qui permet de représenter les mots absents du lexique fasttext avec un token spécial `<unk>`. Ce token reçoit un embedding qui doit être appris. Pour cela, lors de l'entraînement, vous pouvez remplacer certains tokens au hasard par le token `<unk>`. La façon la plus simple/efficace de faire cela est de remplacer certains mots rares directement dans les données lors du pré-traitement (par exemple, une occurrence sur deux des mots apparaissant deux fois). Alternativement, vous pouvez faire du masquage dynamique aléatoirement dans le modèle, lors que vous passez les embeddings vers la couche GRU.

[ ]:

1.6 Pour aller plus loin

- Calculer le F-score sur les segments biens reconnus dans la fonction d'évaluation des performances sur l'ensemble de validation. Cela demande de reconnaître des séquences `B` et `I` et de les considérer comme une seule prédiction.
- Quel est l'impact des hyper-paramètres sur le modèle ?
- Essayez d'ajouter des features morphologiques à ce modèle en concatenant les embeddings de mots avec les sorties d'un CNN sur les caractères du mot (couche de convolution + couche de pooling).