CSCI544 HW4

Yuhang Xiao - 6913860906 - yxiao776@usc.edu

Environment requirements:

- **Python 3.12.1**
- **Numpy**
- **Pandas**
- **torch**
- **tqdm**

Report

Simple Bidirectional LSTM model

Implemented and trained Bi-LSTM model according to the requirements.

Validation statistics on dev data:

- **precision: 85.01%**
- **recall: 76.52%**
- **F1 score: 80.54%**

The F1 score is reasonable compared to the reference 77%.

Using GloVe word embeddings

Used the pretrained GloVe embeddings to help train the model. Because Glove is case-insensitive, I didn't freeze the embeddings and allowed the fine-tuning because NER is case-sensitive.

Validation statistics on dev data:

- **precision: 90.67%**
- **recall: 90.95%**
- **F1 score: 90.81%**

The F1 score is reasonable compared to the reference 88%.

Bonus: LSTM-CNN model

To add the char info into the model. I padded each char of each word to conform to the longest sequence and longest word in each batch. I only added one single cnn layer with a context window of 3 to perform convolutions on chars of each word. Then, a maxpool is utilized to squeeze the word_len dim and let each word's char embeddings could be concatenated to the end of the word embeddings in the embedding_dim dimension. The concatenated features could then be feeded into LSTM to make final predictions.

Validation statistics on dev data:

- **precision: 91.18%**
- **recall: 92.68%**
- **F1 score: 91.92%**

By adding the char info, the performance of the model is improved on all three metrics.

In [1]:

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
```

Data Processing

In [2]:

```
def read_data(file_path, test=False):
    sents = []
    tags = []
    with open(file_path, 'r') as f:
        for line in f.readlines():
            try:
                if test:
                    id, word = line.strip().split(' ')
                    if id == '1':
                        sents.append(sent)
                        sent = []
                    sent.append(word)
                else:
                    id, word, pred = line.strip().split(' ')
                    if id == '1':
                        sents.append(sent)
                        tags.append(tag)
                        sent = []
                        tag = []
                    sent.append(word)
                    tag.append(pred)
            except Exception as e:
                if line.strip() == '':
                    continue
                if isinstance(e, UnboundLocalError):
                    sent = [word]
                    tag = [pred] if not test else []
                    continue
                raise e
    return sents, tags
```

Create Vocabulary

In [3]:

```
sents_train, tags_train = read_data('data/train')
sents_dev, tags_dev = read_data('data/dev')
word_id = {}
word_id['<pad>'] = 0
word_id['<unk>'] = 1
tag_id = {'<pad>': 0}
word_lookup = {0: '<pad>', 1: '<unk>'}
tag_lookup = {0: '<pad>'}
sents = sents_train + sents_dev
tags = tags_train + tags_dev
for i in range(len(sents)):
    for j in range(len(sents[i])):
        word = sents[i][j]
        tag = tags[i][j]
        if word not in word_id:
            word_id[word] = len(word_id)
            word_lookup[word_id[word]] = word
        if tag not in tag_id:
            tag_id[tag] = len(tag_id)
            tag_lookup[tag_id[tag]] = tag
```

Tokenizer

In [4]:

```
def tokenize(sent, word_id):
    tokenized_sent = []
    for word in sent:
        if word in word_id:
            tokenized_sent.append(word_id[word])
        else:
            tokenized_sent.append(word_id['<unk>'])
    return tokenized_sent
```

Collate Function

In [5]:

```
def collate_fn(batch):
    from torch.nn.utils.rnn import pad_sequence
    if isinstance(batch[0], tuple):
        # batch.sort(key=lambda x: len(x[0]), reverse=True)
        sents, tags = zip(*batch)
        lengths = [len(sent) for sent in sents]
        sents = pad_sequence(sents, batch_first=True, padding_value=word_id['<pad>']).long()
        tags = pad_sequence(tags, batch_first=True).long()
        return sents, torch.LongTensor(lengths), tags
    else:
        # batch.sort(key=lambda x: len(x), reverse=True)
        sents = batch
        lengths = [len(sent) for sent in sents]
        sents = pad_sequence(sents, batch_first=True, padding_value=word_id['<pad>']).long()
        return sents, torch.LongTensor(lengths)
```

DataSet

In [6]:

```
class NERDataset(Dataset):
    def __init__(self, sents, tags=None, test=False):
        self.sents = sents
        self.tags = tags
        self.test = test

    def __len__(self):
        return len(self.sents)

    def __getitem__(self, idx):
        if self.test:
            return torch.LongTensor(tokenize(self.sents[idx], word_id))
        else:
            return torch.LongTensor(tokenize(self.sents[idx], word_id)), torch.LongTensor([tag_id[
tag] for tag in self.tags[idx]])
```

Trainer

In [7]:

```
from tqdm import tqdm
class Trainer():
    def __init__(self, model, dataloader, lr_scheduler, optimizer, criterion, epoches=30, device=N
one, freq=10):
        self.model = model
        self.dataloader = dataloader
        self.lr_scheduler = lr_scheduler
        self.optimizer = optimizer
        self.criterion = criterion
        self.epoches = epoches
        self.device = device if device else 'cuda' if torch.cuda.is_available() else 'cpu'
        self.freq = freq

    def train(self):
        self.model.to(self.device)
        self.criterion.to(self.device)
        self.model.train()
        for epoch in tqdm(range(self.epoches)):
            total_loss = 0
            for sents, lengths, tags in self.dataloader:
                sents, lengths, tags = sents.to(self.device, non_blocking=True), lengths, tags.to(
self.device, non_blocking=True)
                self.optimizer.zero_grad()
                outputs = self.model(sents, lengths)
                outputs = outputs.view(-1, outputs.shape[-1])
                tags = tags.view(-1)
                loss = self.criterion(outputs, tags)
                loss.backward()
                self.optimizer.step()
                total_loss += loss.item()
            self.lr_scheduler.step()
            if epoch % self.freq == 0:
                print(f'Epoch {epoch+1}/{self.epoches}, Loss: {total_loss/len(self.dataloader)}')

    def val(self, dataloader, name):
        self.model.to(self.device)
        self.model.eval()
        with open(name, 'w') as f:
            with torch.no_grad():
                for sents, lengths, tags in dataloader:
                    sents, lengths, tags = sents.to(self.device, non_blocking=True), lengths, tags.to(self.device, non_blocking=True)
                    outputs = self.model(sents, lengths)
                    _, predicted = torch.max(outputs, 2)
                    sents = sents.cpu().numpy()
                    predicted = predicted.cpu().numpy()
                    tags = tags.cpu().numpy()
                    lengths = lengths.cpu().numpy()
                    for i in range(len(sents)):
                        for j in range(lengths[i]):
                            f.write(f'{j+1} {word_lookup[sents[i][j]]} {tag_lookup[tags[i][j]]} {tag_lookup[predicted[i][j]]}\n')
                        f.write('\n')

    def test(self, dataloader, name):
        self.model.to(self.device)
        self.model.eval()
        with open(name, 'w') as f:
            with torch.no_grad():
                for sents, lengths in dataloader:
                    sents, lengths = sents.to(self.device, non_blocking=True), lengths
                    outputs = self.model(sents, lengths)
                    _, predicted = torch.max(outputs, 2)
                    sents = sents.cpu().numpy()
                    predicted = predicted.cpu().numpy()
                    lengths = lengths.cpu().numpy()
                    for i in range(len(sents)):
                        for j in range(lengths[i]):
                            f.write(f'{j+1} {word_lookup[sents[i][j]]} {tag_lookup[predicted[i][j]]}\n')
                        f.write('\n')
```

Simple Bidirectional LSTM model

In [8]:

```
class BiLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, target_size, num_layer=1, dropout=0.33, init_embedding=None):
        super(BiLSTM, self).__init__()
        if init_embedding is not None:
            self.word_embeddings = nn.Embedding.from_pretrained(init_embedding, padding_idx=word_id['<pad>'], freeze=False)
        else:
            self.word_embeddings = nn.Embedding(vocab_size, embedding_dim, padding_idx=word_id['<pad>'])
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=num_layer, bidirectional=True, batch_first=True, dropout=0 if num_layer == 1 else dropout)
        self.dropout = nn.Dropout(dropout) if num_layer == 1 else nn.Identity()
        num_direction = 2 if self.lstm.bidirectional else 1
        self.linear = nn.Linear(hidden_dim * num_direction, output_dim)
        self.ELU = nn.ELU()
        self.classifier = nn.Linear(output_dim, target_size)

    def forward(self, sentence, lengths):
        embeds = self.word_embeddings(sentence)
        packed = nn.utils.rnn.pack_padded_sequence(embeds, lengths, batch_first=True, enforce_sorted=False)
        lstm_out, _ = self.lstm(packed)
        lstm_out, _ = nn.utils.rnn.pad_packed_sequence(lstm_out, batch_first=True)
        lstm_out = self.dropout(lstm_out)
        output = self.linear(lstm_out)
        output = self.ELU(output)
        tag_space = self.classifier(output)
        return tag_space
```

Train

In [9]:

```
epoches = 50
batch_size = 128
torch.manual_seed(0)

NERDataset_train = NERDataset(sents_train, tags_train)
dataloader_train = DataLoader(NERDataset_train, batch_size=batch_size, shuffle=True, collate_fn=collate_fn, num_workers=8)

device = 'cuda' if torch.cuda.is_available() else 'cpu'

model = BiLSTM(len(word_id), 100, 256, 128, len(tag_id), num_layer=1, dropout=0.33)
criterion = nn.CrossEntropyLoss(ignore_index=tag_id['<pad>'])
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1, momentum=0.99, weight_decay=1e-4)
lr_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epoches, eta_min=1e-2)

trainer = Trainer(model, dataloader_train, lr_scheduler, optimizer, criterion, epoches=epoches, device=device)
trainer.train()
torch.save(model, 'blstm1.pt')
```

```
  2%|▏         | 1/50 [00:03<03:02,  3.73s/it]
Epoch 1/50, Loss: 0.944644365270259
 22%|██        | 11/50 [00:38<02:17,  3.52s/it]
Epoch 11/50, Loss: 0.11501487896982897
 42%|████      | 21/50 [01:14<01:42,  3.55s/it]
Epoch 21/50, Loss: 0.018057272090750226
 62%|██████    | 31/50 [01:52<01:13,  3.89s/it]
Epoch 31/50, Loss: 0.0054055079186366775
 82%|████████  | 41/50 [02:32<00:35,  3.94s/it]
Epoch 41/50, Loss: 0.003789166355568726
100%|██████████| 50/50 [03:07<00:00,  3.76s/it]
```

Dev validation

In [10]:

```
NERDataset_dev = NERDataset(sents_dev, tags_dev)
dataloader_dev = DataLoader(NERDataset_dev, batch_size=batch_size, shuffle=False, collate_fn=collate_fn, num_workers=8)
trainer.val(dataloader_dev, 'dev1.out')
!perl conll03eval < dev1.out
```

```
processed 51577 tokens with 5942 phrases; found: 5349 phrases; correct: 4547.
accuracy:  96.11%; precision:  85.01%; recall:  76.52%; FB1:  80.54
              LOC: precision:  92.63%; recall:  84.21%; FB1:  88.22  1670
             MISC: precision:  86.12%; recall:  78.09%; FB1:  81.91  836
              ORG: precision:  78.35%; recall:  72.86%; FB1:  75.50  1247
              PER: precision:  81.64%; recall:  70.74%; FB1:  75.80  1596
```

Test

In [11]:

```
sents_test, _ = read_data('data/test', test=True)
NERDataset_test = NERDataset(sents_test, test=True)
dataloader_test = DataLoader(NERDataset_test, batch_size=batch_size, shuffle=False, collate_fn=collate_fn, num_workers=8)
trainer.test(dataloader_test, 'test1.out')
```

Using GloVe word embeddings

load GloVe weights

In [12]:

```
glove = pd.read_csv('./glove.6B.100d.gz', sep=" ", quoting=3, header=None, index_col=0)
glove_dict = {k: v.values for k, v in glove.T.items()}
glove_mat = np.array([glove_dict[k] for k in glove_dict])
glove_dict['<pad>'] = np.zeros(100)
glove_dict['<unk>'] = np.mean(glove_mat, axis=0)
init_embedding = torch.tensor(np.array([glove_dict[word_lookup[i]] if word_lookup[i] in glove_dict else
                                glove_dict[word_lookup[i].lower()] + 5e-3 if word_lookup[i].lower() in glove_dict else glove_dict['<unk>']
                                for i in range(len(word_id))]))
```

Train

In [13]:

```
epoches = 50
batch_size = 128
torch.manual_seed(0)

NERDataset_train = NERDataset(sents_train, tags_train)
dataloader_train = DataLoader(NERDataset_train, batch_size=batch_size, shuffle=True, collate_fn=collate_fn, num_workers=8)

device = 'cuda' if torch.cuda.is_available() else 'cpu'

model = BiLSTM(len(word_id), 100, 256, 128, len(tag_id), num_layer=1, dropout=0.33, init_embedding=init_embedding.float())
criterion = nn.CrossEntropyLoss(ignore_index=tag_id['<pad>'], label_smoothing=0.1)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1, momentum=0.99, nesterov=True)
lr_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epoches, eta_min=8e-2)

trainer = Trainer(model, dataloader_train, lr_scheduler, optimizer, criterion, epoches=epoches, device=device)
trainer.train()
torch.save(model, 'blstm2.pt')
```

```
  0%|          | 0/50 [00:00<?, ?it/s]  2%|▏         | 1/50 [00:04<03:22,  4.14s/it]
Epoch 1/50, Loss: 0.9155444810956211
 22%|██▏       | 11/50 [00:45<02:40,  4.10s/it]
Epoch 11/50, Loss: 0.5344660595311956
 42%|████▏     | 21/50 [01:26<01:59,  4.11s/it]
Epoch 21/50, Loss: 0.5134784089306653
 62%|██████▏   | 31/50 [02:07<01:18,  4.12s/it]
Epoch 31/50, Loss: 0.5075996149394472
 82%|████████▏ | 41/50 [02:48<00:36,  4.07s/it]
Epoch 41/50, Loss: 0.504991847074638
100%|██████████| 50/50 [03:23<00:00,  4.06s/it]
```

Dev validation

In [14]:

```
NERDataset_dev = NERDataset(sents_dev, tags_dev)
dataloader_dev = DataLoader(NERDataset_dev, batch_size=batch_size, shuffle=False, collate_fn=collate_fn, num_workers=8)
trainer.val(dataloader_dev, 'dev2.out')
!perl conll03eval < dev2.out
```

```
processed 51577 tokens with 5942 phrases; found: 5960 phrases; correct: 5404.
accuracy:  98.23%; precision:  90.67%; recall:  90.95%; FB1:  90.81
              LOC: precision:  93.97%; recall:  94.99%; FB1:  94.48  1857
```

```
             LOC: precision:  93.97%; recall:  94.99%; FB1:  94.48  1837
            MISC: precision:  85.05%; recall:  83.95%; FB1:  84.50  910
             ORG: precision:  84.88%; recall:  86.20%; FB1:  85.53  1362
             PER: precision:  94.43%; recall:  93.87%; FB1:  94.15  1831
```

Test

In [15]:

```
sents_test, _ = read_data('data/test', test=True)
NERDataset_test = NERDataset(sents_test, test=True)
dataloader_test = DataLoader(NERDataset_test, batch_size=batch_size, shuffle=False,
collate_fn=collate_fn, num_workers=8)
trainer.test(dataloader_test, 'test2.out')
```

LSTM-CNN model

create character vocabulary

In [16]:

```
char_id = {}
char_id['<pad>'] = 0
char_id['<unk>'] = 1
for i in range(len(sents)):
    for j in range(len(sents[i])):
        for c in sents[i][j]:
            if c not in char_id:
                char_id[c] = len(char_id)
```

Char Dataset

In [17]:

```
def tokenize_char(sent, char_id):
    tokenized_char = []
    for word in sent:
        word_char = []
        for c in word:
            if c in char_id:
                word_char.append(char_id[c])
            else:
                word_char.append(char_id['<unk>'])
        word_char = torch.LongTensor(word_char)
        tokenized_char.append(word_char)
    return tokenized_char
```

In [18]:

```
class CharDataset(Dataset):
    def __init__(self, sents, tags=None, test=False):
        self.sents = sents
        self.tags = tags
        self.test = test

    def __len__(self):
        return len(self.sents)

    def __getitem__(self, idx):
        from torch.nn.utils.rnn import pad_sequence
        if self.test:
            return torch.LongTensor(tokenize(self.sents[idx], word_id)), \
                torch.LongTensor(pad_sequence(tokenize_char(self.sents[idx], char_id), batch_first
=True, padding_value=char_id['<pad>']))
        else:
            return torch.LongTensor(tokenize(self.sents[idx], word_id)), torch.LongTensor([tag_id[
tag] for tag in self.tags[idx]]), \
                torch.LongTensor(pad_sequence(tokenize_char(self.sents[idx], char_id), batch_first
=True, padding_value=char_id['<pad>']))
```

In [19]:

```
def collate_fn_char(batch):
    from torch.nn.utils.rnn import pad_sequence
    import torch.nn.functional as F
    if isinstance(batch[0], tuple) and len(batch[0]) == 3:
        sents, tags, chars = zip(*batch)
        lengths = [len(sent) for sent in sents]
        sents = pad_sequence(sents, batch_first=True, padding_value=word_id['<pad>']).long()
        tags = pad_sequence(tags, batch_first=True).long()
        max_word_len = max(char.shape[1] for char in chars)
        max_seq_len = sents.shape[1]
        chars = torch.stack([F.pad(char, (0, max_word_len - char.shape[1], 0, max_seq_len - char.shape[0]), value=char_id['<pad>']) for char in chars])
        return sents, torch.LongTensor(lengths), tags, chars
    else:
        sents, chars = zip(*batch)
        lengths = [len(sent) for sent in sents]
        sents = pad_sequence(sents, batch_first=True, padding_value=word_id['<pad>']).long()
        max_word_len = max(char.shape[1] for char in chars)
        max_seq_len = sents.shape[1]
        chars = torch.stack([F.pad(char, (0, max_word_len - char.shape[1], 0, max_seq_len - char.shape[0]), value=char_id['<pad>']) for char in chars])
        return sents, torch.LongTensor(lengths), chars
```

Prepare model

In [20]:

```
class CharBiLSTM(BiLSTM):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, target_size, char_vocab_size, char_embedding_dim, num_layer=1, dropout=0.33, init_embedding=None):
        super().__init__(vocab_size, embedding_dim, hidden_dim, output_dim, target_size, num_layer, dropout, init_embedding)
        self.char_embeddings = nn.Embedding(char_vocab_size, char_embedding_dim, padding_idx=char_id['<pad>'])
        self.char_cnn = nn.Conv1d(char_embedding_dim, char_embedding_dim, 3, padding=1)
        self.char_maxpool = nn.AdaptiveMaxPool1d(1)
        self.lstm = nn.LSTM(embedding_dim + char_embedding_dim, hidden_dim, num_layers=num_layer,
                            bidirectional=True, batch_first=True, dropout=0 if num_layer == 1 else dropout)

    def forward(self, sentence, lengths, chars):
        embeds = self.word_embeddings(sentence)
        char_embeds = self.char_embeddings(chars)
        b, s, w, c = char_embeds.shape
        char_embeds = char_embeds.view(b*s, w, c).permute(0, 2, 1)
        char_embeds = self.char_cnn(char_embeds)
        char_embeds = self.char_maxpool(char_embeds).squeeze(-1)
        char_embeds = char_embeds.view(b, s, -1)
        embeds = torch.cat([embeds, char_embeds], dim=-1)
        packed = nn.utils.rnn.pack_padded_sequence(embeds, lengths, batch_first=True, enforce_sorted=False)
        lstm_out, _ = self.lstm(packed)
        lstm_out, _ = nn.utils.rnn.pad_packed_sequence(lstm_out, batch_first=True)
        lstm_out = self.dropout(lstm_out)
        output = self.linear(lstm_out)
        output = self.ELU(output)
        tag_space = self.classifier(output)
        return tag_space
```

Char trainer

In [21]:

```
from tqdm import tqdm
class CharTrainer():
    def __init__(self, model, dataloader, lr_scheduler, optimizer, criterion, epoches=30, device=None, freq=10):
        self.model = model
        self.dataloader = dataloader
        self.lr_scheduler = lr_scheduler
        self.optimizer = optimizer
        self.criterion = criterion
        self.epoches = epoches
        self.device = device if device else 'cuda' if torch.cuda.is_available() else 'cpu'
        self.freq = freq

    def train(self):
        self.model.to(self.device)
        self.criterion.to(self.device)
        self.model.train()
        for epoch in tqdm(range(self.epoches)):
            total_loss = 0
            for sents, lengths, tags, chars in self.dataloader:
                sents, lengths, tags, chars = sents.to(self.device, non_blocking=True), lengths, \
                    tags.to(self.device, non_blocking=True), chars.to(self.device, non_blocking=Tr
ue)
                self.optimizer.zero_grad()
                outputs = self.model(sents, lengths, chars)
                outputs = outputs.view(-1, outputs.shape[-1])
                tags = tags.view(-1)
                loss = self.criterion(outputs, tags)
                loss.backward()
                self.optimizer.step()
                total_loss += loss.item()
            self.lr_scheduler.step()
            if epoch % self.freq == 0:
                print(f'Epoch {epoch+1}/{self.epoches}, Loss: {total_loss/len(self.dataloader)}')

    def val(self, dataloader, name):
        self.model.to(self.device)
        self.model.eval()
        with open(name, 'w') as f:
            with torch.no_grad():
                for sents, lengths, tags, chars in dataloader:
                    sents, lengths, tags, chars = sents.to(self.device, non_blocking=True), length
s, \
                        tags.to(self.device, non_blocking=True), chars.to(self.device,
non_blocking=True)
                    outputs = self.model(sents, lengths, chars)
                    _, predicted = torch.max(outputs, 2)
                    sents = sents.cpu().numpy()
                    predicted = predicted.cpu().numpy()
                    tags = tags.cpu().numpy()
                    lengths = lengths.cpu().numpy()
                    for i in range(len(sents)):
                        for j in range(lengths[i]):
                            f.write(f'{j+1} {word_lookup[sents[i][j]]} {tag_lookup[tags[i][j]]} {t
ag_lookup[predicted[i][j]]}\n')
                        f.write('\n')

    def test(self, dataloader, name):
        self.model.to(self.device)
        self.model.eval()
        with open(name, 'w') as f:
            with torch.no_grad():
                for sents, lengths, chars in dataloader:
                    sents, lengths, chars = sents.to(self.device, non_blocking=True), lengths, cha
rs.to(self.device, non_blocking=True)
                    outputs = self.model(sents, lengths, chars)
                    _, predicted = torch.max(outputs, 2)
                    sents = sents.cpu().numpy()
                    predicted = predicted.cpu().numpy()
                    lengths = lengths.cpu().numpy()
                    for i in range(len(sents)):
                        for j in range(lengths[i]):
                            f.write(f'{j+1} {word_lookup[sents[i][j]]} {tag_lookup[predicted[i][j]
]}\n')
                        f.write('\n')
```

Train

In [22]:

```
epoches = 50
batch_size = 128
torch.manual_seed(0)

CharDataset_train = CharDataset(sents_train, tags_train)
dataloader_train = DataLoader(CharDataset_train, batch_size=batch_size, shuffle=True, collate_fn=c
ollate_fn_char, num_workers=8)

device = 'cuda' if torch.cuda.is_available() else 'cpu'

model = CharBiLSTM(len(word_id), 100, 256, 128, len(tag_id), len(char_id), 30, num_layer=1, dropou
t=0.33, init_embedding=init_embedding.float())
criterion = nn.CrossEntropyLoss(ignore_index=tag_id['<pad>'], label_smoothing=0.1)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1, momentum=0.99, nesterov=True)
lr_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epoches, eta_min=8e-2)

trainer = CharTrainer(model, dataloader_train, lr_scheduler, optimizer, criterion, epoches=epoches
, device=device)
trainer.train()
torch.save(model, 'blstm-cnn.pt')
```

```
  0%|          | 0/50 [00:00<?, ?it/s]  2%|▏         | 1/50 [00:06<04:58,  6.10s/it]
```

```
Epoch 1/50, Loss: 0.8771523373611902
```

```
 22%|██        | 11/50 [00:54<03:09,  4.86s/it]
```

```
Epoch 11/50, Loss: 0.5253389098886716
```

```
 42%|████      | 21/50 [01:43<02:21,  4.87s/it]
```

```
Epoch 21/50, Loss: 0.5106210188340332
```

```
 62%|██████    | 31/50 [02:32<01:32,  4.86s/it]
```

```
Epoch 31/50, Loss: 0.506106627694631
```

```
 82%|████████  | 41/50 [03:21<00:43,  4.87s/it]
```

```
Epoch 41/50, Loss: 0.5042909095853062
```

```
100%|██████████| 50/50 [04:05<00:00,  4.90s/it]
```

DEV validation

In [23]:

```
CharDataset_dev = CharDataset(sents_dev, tags_dev)
dataloader_dev = DataLoader(CharDataset_dev, batch_size=batch_size, shuffle=False,
collate_fn=collate_fn_char, num_workers=8)
trainer.val(dataloader_dev, 'dev-cnn.out')
!perl conll03eval < dev-cnn.out
```

```
processed 51577 tokens with 5942 phrases; found: 6040 phrases; correct: 5507.
accuracy:  98.64%; precision:  91.18%; recall:  92.68%; FB1:  91.92
              LOC: precision:  95.05%; recall:  95.10%; FB1:  95.07  1838
             MISC: precision:  86.45%; recall:  87.20%; FB1:  86.83  930
              ORG: precision:  83.84%; recall:  88.22%; FB1:  85.97  1411
              PER: precision:  95.27%; recall:  96.25%; FB1:  95.76  1861
```

Test

In [24]:

```
sents_test, _ = read_data('data/test', test=True)
CharDataset_test = CharDataset(sents_test, test=True)
dataloader_test = DataLoader(CharDataset_test, batch_size=batch_size, shuffle=False, collate_fn=co
llate_fn_char, num_workers=8)
trainer.test(dataloader_test, 'pred')
```