CSCI544: Homework Assignment No. 4

Submitted by:

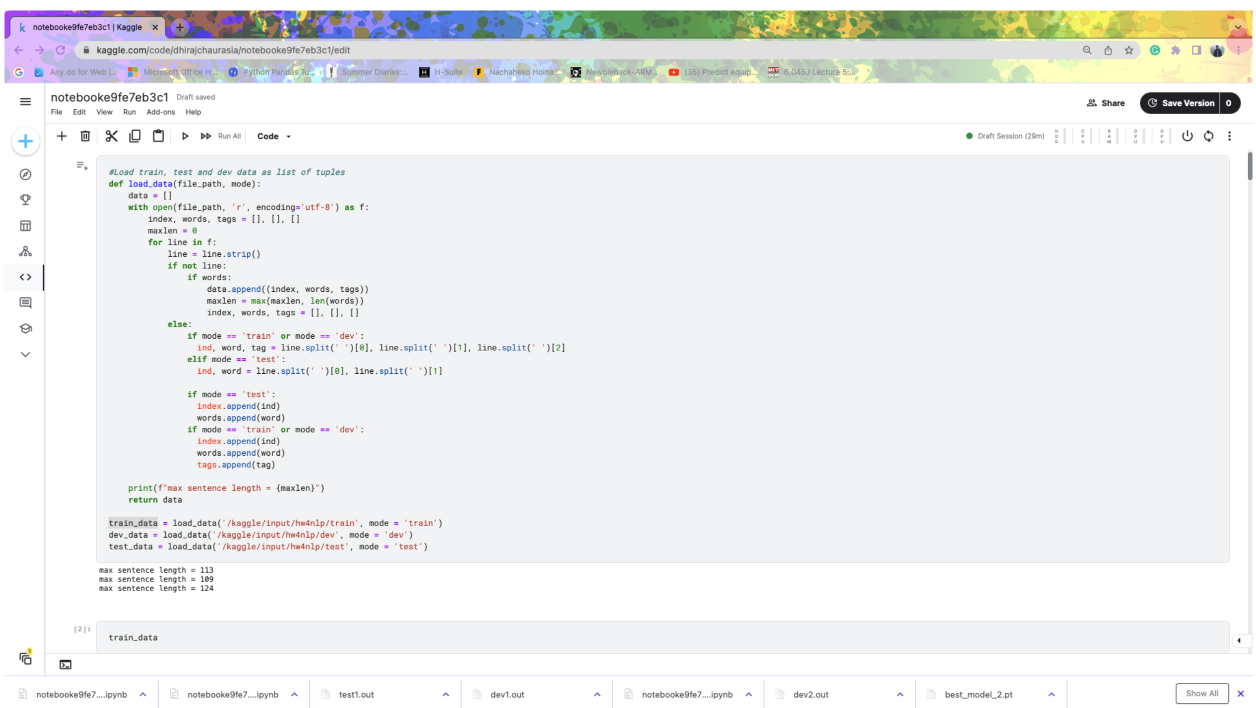
Name: Dhiraj Chaurasia
USC ID#: 1556515687

Task 1: Simple Bidirectional LSTM model (40 points)

Note: Please refer to the notebook attached in the submission

Steps:

1. Load train, test and dev data as list of tuples




```python
#Load train, test and dev data as list of tuples
def load_data(file_path, mode):
    data = []
    with open(file_path, 'r', encoding='utf-8') as f:
        index, words, tags = [], [], []
        maxlen = 0
        for line in f:
            line = line.strip()
            if not line:
                if words:
                    data.append((index, words, tags))
                    maxlen = max(maxlen, len(words))
                    index, words, tags = [], [], []
            else:
                if mode == 'train' or mode == 'dev':
                  ind, word, tag = line.split(' ')[0], line.split(' ')[1], line.split(' ')[2]
                elif mode == 'test':
                  ind, word = line.split(' ')[0], line.split(' ')[1]

                if mode == 'test':
                  index.append(ind)
                  words.append(word)
                if mode == 'train' or mode == 'dev':
                  index.append(ind)
                  words.append(word)
                  tags.append(tag)

    print(f"max sentence length = {maxlen}")
    return data

train_data = load_data('/kaggle/input/hw4nlp/train', mode = 'train')
dev_data = load_data('/kaggle/input/hw4nlp/dev', mode = 'dev')
test_data = load_data('/kaggle/input/hw4nlp/test', mode = 'test')
```

```
max sentence length = 113
max sentence length = 109
max sentence length = 124
```

```python
[2]: train_data
```

[2]:

```
train_data
```

[2]:

```
[(['1', '2', '3', '4', '5', '6', '7', '8', '9'],
  ['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb', '.'],
  ['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O', 'O']),
 (['1', '2'], ['Peter', 'Blackburn'], ['B-PER', 'I-PER']),
 (['1', '2'], ['BRUSSELS', '1996-08-22'], ['B-LOC', 'O']),
 (['1',
   '2',
   '3',
   '4',
```

2. Build vocabulary and word<->tag maps

```python
# Build vocabulary and word<->tag maps

from collections import Counter
def build_vocab(data):
    word_counts = Counter(word for _, sentence, _ in data for word in sentence)
    filtered_dict = {key: value for key, value in word_counts.items()}
    vocabulary = ['<pad>', '<unk>'] + sorted(filtered_dict)
    word2idx = {word: idx for idx, word in enumerate(vocabulary)}
    return vocabulary, word2idx

def build_tag_map(data):
    tags = set(tag for _,_, tags in data for tag in tags)
    # tags.add('<pad>')
    tag2idx = {tag: idx for idx, tag in enumerate(sorted(tags))}
    return tag2idx

vocabulary, word2idx = build_vocab(train_data + dev_data + test_data)
tag2idx = build_tag_map(train_data + dev_data)
```

[5]:

```
vocabulary[:5]
```

[5]:

```
['<pad>', '<unk>', '!', '"', '$']
```

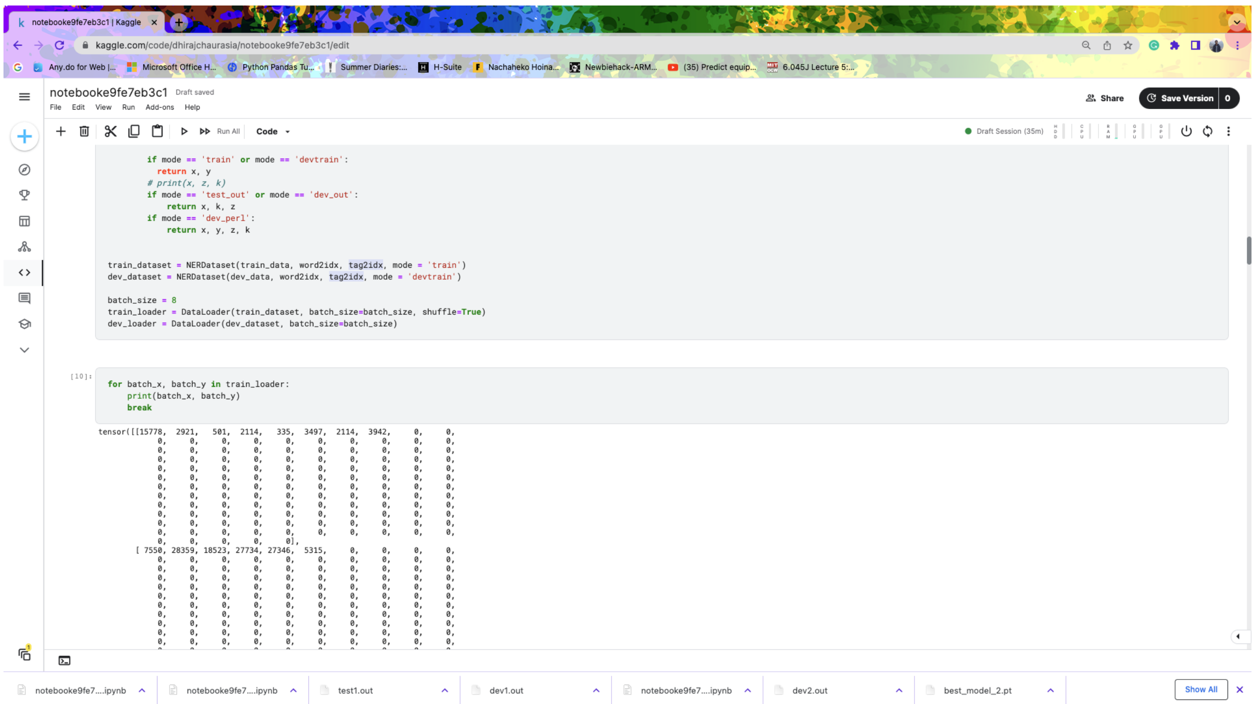
[8]:

```
word2idx
```

[8]:

```
{'<pad>': 0,
 '<unk>': 1,
 '!': 2,
 '"': 3,
 '$': 4,
 '%': 5,
 '&': 6,
 "'": 7,
 "'S": 8,
 "'d": 9,
 "'ll": 10,
 "'m": 11,
```

3. Make a DataLoader that takes raw data -> tokenizes -> pads -> and returns batches of data.



4. Code the model
   Note: Speed up training time by using pack_padded_sequence and pad_packed_sequence

notebooke9fe7eb3c1 Draft saved


```python
#Define Model

import torch.nn as nn
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
import torch.nn.init as init


class BLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, dropout, pretrained_embedding = None):
        super(BLSTM, self).__init__()
        if pretrained_embedding:
            self.embedding = pretrained_embedding
        else:
            self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
#          init.xavier_uniform_(self.embedding.weight)
        self.blstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=1, bidirectional=True, batch_first=True)
        self.dropout1 = nn.Dropout(dropout)
        self.linear1 = nn.Linear(hidden_dim * 2, 128)
        self.activation = nn.ELU()
        self.classifier = nn.Sequential(
            nn.Linear(128, 256),
            nn.Tanh(),
            nn.Dropout(0.2),
            nn.Linear(256, output_dim)
        )

    def forward(self, x):
        # print(x.shape)
        embedded = self.embedding(x)
        seq_lengths = torch.count_nonzero(x, dim=1).cpu()
        x = pack_padded_sequence(embedded, seq_lengths, batch_first=True, enforce_sorted=False)
        x, _ = self.blstm(x)
        x, unpacked_len = pad_packed_sequence(x, batch_first=True)
        x = self.dropout1(x)
        x = self.linear1(x)
        x = self.activation(x)
        x = self.classifier(x)
        return x
```

5. Utility function to monitor the training process

```
# piecewise accuracy
def accuracy(outputs, labels):
    acc = 0
    count = 0
    for i in range(outputs.shape[0]):
        sentence_pred = outputs[i]
        for j, word in enumerate(sentence_pred):
            word_pred = torch.argmax(word).item()
            label = labels[i][j].item()
            if label == -1:
                continue
            count += 1
            if word_pred == label:
                acc += 1
    return acc/count

#evaluate function for dev test during training
def evaluate(model, criterion, dataloader, device = 'cuda'):
  with torch.no_grad():
    dev_loss, dev_acc, dev_f1 = 0.0, 0.0, 0.0
    for batch_x, batch_y in tqdm(dataloader):
      batch_x = batch_x.to(device)
      batch_y = batch_y.to(device)
      outputs = model(batch_x)
      seq_lengths = torch.count_nonzero(batch_x, dim=1).to('cpu')
      packed_y = pack_padded_sequence(batch_y, seq_lengths, batch_first=True, enforce_sorted=False)
      unpacked_y, unpacked_len = pad_packed_sequence(packed_y, batch_first=True, padding_value=-1)
      unpacked_y = unpacked_y.to(device)
      loss = criterion(outputs.permute(0, 2, 1), unpacked_y)
      dev_loss += loss.item()
      out_for_f1 = torch.argmax(outputs, dim = -1)
      mask = (unpacked_y >= 0)
      f1 = f1_score(out_for_f1[mask].cpu(), unpacked_y[mask].cpu(), average='weighted')
      #-->costly operation, uncomment to see accuracy
#       acc = accuracy(outputs, batch_y)
#       dev_acc += acc
      dev_f1 += f1
    dev_loss /= len(dataloader)
    dev_acc /= len(dataloader)
    dev_f1 /= len(dataloader)
```

Piecewise accuracy measures label_hits/total_data_points excluding padding.

In evaluate model, I also calculate the F1 score (after masking padding).

6. Compute class weights to handle class imbalance

```
from sklearn.utils.class_weight import compute_class_weight

def get_class_weights(data):
    all_y = []
    for data in data:
        all_y.extend(data[2])

    class_weights = compute_class_weight(
                                        class_weight = "balanced",
                                        classes = np.unique(all_y),
                                        y = all_y
                                    )
    class_weights=torch.tensor(class_weights,dtype=torch.float).to('cuda')
    return class_weights

class_weights = get_class_weights(train_data + dev_data)
```

```
# class_weights -= torch.min(class_weights)
class_weights
```

```
tensor([ 3.1704,  6.5276,  3.7145,  3.3713, 20.1275, 18.9609,  6.3884,  4.8775,
         0.1333], device='cuda:0')
```

7. Define needed stuffs

```
from tqdm import tqdm
from sklearn.metrics import f1_score
from torchmetrics.functional.classification import multiclass_f1_score
from torch.optim.lr_scheduler import MultiStepLR


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BLSTM(len(vocabulary), embedding_dim=100, hidden_dim=256, output_dim=len(tag2idx), dropout=0.33).to(device)
# optimizer = torch.optim.Adam(model.parameters(), lr = 0.001, weight_decay = 1e-5, eps=1e-08, betas = (0.9, 0.999))
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9, weight_decay = 1e-3)
# scheduler = MultiStepLR(optimizer, milestones=[3,5,7,9], gamma=0.1)
# scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.5, max_lr=1.2, step_size_up=20, step_size_down=None, mode='triangular', gamma=1.0)
# scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', min_lr=1, verbose=True)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',factor=0.55, patience = 3, threshold=0.1, verbose=True, min_lr=5e-4)
# scheduler = torch.optim.lr_scheduler.LinearLR(optimizer, start_factor=0.5, total_iters=20)

# class_weights = torch.tensor([1,1,1,1,1,1,1,1,0.01], dtype=torch.float).to(device)
criterion = nn.CrossEntropyLoss(ignore_index=-1, reduction='mean', weight=class_weights)
```

Note: Tried various optimizers and schedulers to get a feel of the loss space. Used SGD optimizer and ReduceLROnPlateau in the end.

8. Train the model

```python
def train(model, train_loader, optimizer, criterion, device, epochs):
    model.train()
    SAVE_PATH = "./best_model.pt"
    best_f1 = -1
    for epoch in range(epochs):
      print(f"Epoch: {epoch}")
      train_loss, train_acc, train_f1 = 0.0, 0.0, 0.0
      for batch_x, batch_y in tqdm(train_loader):
          batch_x = batch_x.to(device)
          batch_y = batch_y.to(device)
          outputs = model(batch_x)

          seq_lengths = torch.count_nonzero(batch_x, dim=1).to('cpu')
          packed_y = pack_padded_sequence(batch_y, seq_lengths, batch_first=True, enforce_sorted=False)
          unpacked_y, unpacked_len = pad_packed_sequence(packed_y, batch_first=True, padding_value = -1)
          unpacked_y = unpacked_y.to(device)

          loss = criterion(outputs.permute(0, 2, 1), unpacked_y)
          train_loss += loss.item()
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()

      train_loss /= len(train_loader)
      val_loss, val_f1 = evaluate(model, criterion, dev_loader)
      if val_f1 > best_f1:
            best_f1 = val_f1
            torch.save(model.state_dict(), SAVE_PATH)
      scheduler.step(val_loss)
      print(f"Average train Loss: {train_loss}")
      print(f"Current Learning Rate: {get_lr(optimizer)}")
      print(f"Best sklearn masked F1: {best_f1}")

    return model

model = train(model, train_loader, optimizer, criterion, device, epochs = 100)
```

```
Epoch: 0
100%|██████████| 1874/1874 [00:18<00:00, 101.18it/s]
100%|██████████| 434/434 [00:02<00:00, 164.55it/s]
Average Dev Loss: 1.4999754266804814
Average Dev accuracy: 0.0
```

I trained the model for 100 epochs to see how it fits. I adjusted the hyperparameters based on several experiments.

Notes

- Training seems to benefit from reducing the learning rate over epochs.
- I use a patience of 3 and penalize the learning rate if the dev loss is not decreasing.
- Although the model seems to saturate for a while but if we let it learn nevertheless for a few more epochs, we observe that the sklearn F1 increases even though dev loss seems stagnant. Then after a few epochs, loss also decreases.
- Use early stopping, save best model and load it later

```
Epoch: 0
100%|██████████| 1874/1874 [00:18<00:00, 101.18it/s]
100%|██████████| 434/434 [00:02<00:00, 164.55it/s]
Average Dev Loss: 1.4999754266804814
Average Dev accuracy: 0.0
Average Dev F1: 0.44007233480329083
Average train Loss: 1.7876404129135697
Current Learning Rate: 0.01
Best sklearn masked F1: 0.44007233480329083
Epoch: 1
100%|██████████| 1874/1874 [00:18<00:00, 103.24it/s]
100%|██████████| 434/434 [00:02<00:00, 168.88it/s]
Average Dev Loss: 1.2480844843634813
Average Dev accuracy: 0.0
Average Dev F1: 0.512941720119775
Average train Loss: 1.3169132836854827
Current Learning Rate: 0.01
Best sklearn masked F1: 0.512941720119775
Epoch: 2
100%|██████████| 1874/1874 [00:18<00:00, 99.47it/s]
100%|██████████| 434/434 [00:02<00:00, 180.12it/s]
Average Dev Loss: 1.130530881613905
Average Dev accuracy: 0.0
Average Dev F1: 0.514520515226967
Average train Loss: 1.0797002766849901
Current Learning Rate: 0.01
Best sklearn masked F1: 0.514520515226967
Epoch: 3
100%|██████████| 1874/1874 [00:18<00:00, 103.54it/s]
100%|██████████| 434/434 [00:02<00:00, 153.28it/s]
Average Dev Loss: 1.064261501354556
Average Dev accuracy: 0.0
Average Dev F1: 0.5735050098465089
Average train Loss: 0.9368670036024319
Current Learning Rate: 0.01
Best sklearn masked F1: 0.5735050098465089
Epoch: 4
100%|██████████| 1874/1874 [00:18<00:00, 103.72it/s]
100%|██████████| 434/434 [00:02<00:00, 176.84it/s]
Average Dev Loss: 1.0281429839779705
Average Dev accuracy: 0.0
Average Dev F1: 0.6522480739744296
Average train Loss: 0.8510956046198322
Current Learning Rate: 0.01
Best sklearn masked F1: 0.6522480739744296
Epoch: 5
100%|██████████| 1874/1874 [00:18<00:00, 101.43it/s]
100%|██████████| 434/434 [00:02<00:00, 157.99it/s]
Average Dev Loss: 1.0095814675401706
Average Dev accuracy: 0.0
Average Dev F1: 0.494183893979238
Average train Loss: 0.7891013945354469
```

```
Average Dev F1: 0.9539465449150528
Average train Loss: 0.025110982044991563
Current Learning Rate: 0.0005
Best sklearn masked F1: 0.9596342079482945
Epoch: 96
100%|██████████| 1874/1874 [00:18<00:00, 103.45it/s]
100%|██████████| 434/434 [00:02<00:00, 160.55it/s]
Average Dev Loss: 0.5426721245561156
Average Dev accuracy: 0.0
Average Dev F1: 0.9574477554482997
Average train Loss: 0.02484619014177336
Current Learning Rate: 0.0005
Best sklearn masked F1: 0.9596342079482945
Epoch: 97
100%|██████████| 1874/1874 [00:18<00:00, 102.49it/s]
100%|██████████| 434/434 [00:02<00:00, 178.45it/s]
Average Dev Loss: 0.5446643792452239
Average Dev accuracy: 0.0
Average Dev F1: 0.9574796125732961
Average train Loss: 0.02480737278372121
Current Learning Rate: 0.0005
Best sklearn masked F1: 0.9596342079482945
Epoch: 98
100%|██████████| 1874/1874 [00:18<00:00, 101.17it/s]
100%|██████████| 434/434 [00:02<00:00, 156.27it/s]
Average Dev Loss: 0.5294855833879762
Average Dev accuracy: 0.0
Average Dev F1: 0.9546081339872722
Average train Loss: 0.024253755565230777
Current Learning Rate: 0.0005
Best sklearn masked F1: 0.9596342079482945
Epoch: 99
100%|██████████| 1874/1874 [00:17<00:00, 104.61it/s]
100%|██████████| 434/434 [00:02<00:00, 160.47it/s]
Average Dev Loss: 0.5564461686929542
Average Dev accuracy: 0.0
Average Dev F1: 0.9567873982440075
Average train Loss: 0.024438474007152624
Current Learning Rate: 0.0005
Best sklearn masked F1: 0.9596342079482945
```

Final Results:

```
3]: !perl conll03eval < dev1.out
```

```
processed 51577 tokens with 5942 phrases; found: 5673 phrases; correct: 4645.
accuracy:  96.06%; precision:  81.88%; recall:  78.17%; FB1:  79.98
              LOC: precision:  90.58%; recall:  85.36%; FB1:  87.89  1731
             MISC: precision:  79.12%; recall:  74.40%; FB1:  76.69  867
              ORG: precision:  73.10%; recall:  73.97%; FB1:  73.54  1357
              PER: precision:  81.43%; recall:  75.95%; FB1:  78.60  1718
```

Task 2: Using GloVe word embeddings (60 points)

Steps:

1) Most of the parts of code including dataloader and vocab remain the same.
2) Load Glove Embeddings

```
[84]:
embeddings_index = {}
with open('glove.6B.100d', 'r', encoding='utf-8') as f:
    for line in tqdm(f):
        values = line.split()
        word = values[0].lower()
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

len(embeddings_index)
```

```
400000it [00:09, 41931.32it/s]
[84… 400000
```

3) Make weight matrix using glove embeddings and initialize the weights of the Embedding Layer using weight matrix

```
def make_weight_matrix(word2idx):
    weights_matrix = np.zeros((len(vocabulary), 100))
    hits = misses = 0
    # Initialize the unk and pad vector randomly using a normal distribution
    unk_weight = np.random.normal(scale=0.8, size=(100,))
    pad_weight = np.random.normal(scale=0.8, size=(100,))
    for word, i in word2idx.items():
        embedding_vector = embeddings_index.get(word.lower())
        if embedding_vector is not None:
            weights_matrix[i] = embedding_vector
            hits += 1
        else:
            misses += 1
            if word == '<pad>':
                weights_matrix[i] = pad_weight
            else:
                weights_matrix[i] = unk_weight
    print(f"Hits: {hits} Misses: {misses} Hit Ratio: {hits/(hits+misses)}")
    return weights_matrix


weights_matrix = make_weight_matrix(word2idx)
embedding_layer = nn.Embedding(len(vocabulary), 100)
embedding_layer.weight.data.copy_(torch.from_numpy(weights_matrix))
embedding_layer.weight.requires_grad = True
```

```
Hits: 26340 Misses: 3952 Hit Ratio: 0.8695365112901096
```

+ Code + Markdown

4) Train the model

```
Epoch: 0
100%|██████████| 1874/1874 [00:18<00:00, 101.88it/s]
100%|██████████| 434/434 [00:02<00:00, 174.71it/s]
Average Dev Loss: 0.7340873273149613
Average Dev accuracy: 0.0
Average Dev F1: 0.7705576421574798
Average train Loss: 1.0544551648954954
Current Learning Rate: 0.01
Best sklearn masked F1: 0.7705576421574798
Epoch: 1
100%|██████████| 1874/1874 [00:18<00:00, 103.64it/s]
100%|██████████| 434/434 [00:02<00:00, 178.20it/s]
Average Dev Loss: 0.541081804717775
Average Dev accuracy: 0.0
Average Dev F1: 0.7525595739373699
Average train Loss: 0.5337999259681304
Current Learning Rate: 0.01
Best sklearn masked F1: 0.7705576421574798
Epoch: 2
100%|██████████| 1874/1874 [00:18<00:00, 102.19it/s]
100%|██████████| 434/434 [00:02<00:00, 170.28it/s]
Average Dev Loss: 0.4140992299245868
Average Dev accuracy: 0.0
Average Dev F1: 0.8172337322708556
Average train Loss: 0.40893494477147674
Current Learning Rate: 0.01
Best sklearn masked F1: 0.8172337322708556
Epoch: 3
100%|██████████| 1874/1874 [00:18<00:00, 103.21it/s]
100%|██████████| 434/434 [00:02<00:00, 158.26it/s]
Average Dev Loss: 0.4684875528070612
Average Dev accuracy: 0.0
Average Dev F1: 0.8982840367921393
Average train Loss: 0.33479627251640964
Current Learning Rate: 0.01
Best sklearn masked F1: 0.8982840367921393
Epoch: 4
100%|██████████| 1874/1874 [00:17<00:00, 104.90it/s]
100%|██████████| 434/434 [00:02<00:00, 169.59it/s]
Average Dev Loss: 0.42387279222208646
Average Dev accuracy: 0.0
Average Dev F1: 0.8954061583715427
Average train Loss: 0.2702637093445821
Current Learning Rate: 0.01
Best sklearn masked F1: 0.8982840367921393
Epoch: 5
100%|██████████| 1874/1874 [00:18<00:00, 102.16it/s]
100%|██████████| 434/434 [00:02<00:00, 179.66it/s]
Average Dev Loss: 0.38185237123618065
Average Dev accuracy: 0.0
Average Dev F1: 0.8988944260272429
Average train Loss: 0.2163824252063645
Current Learning Rate: 0.01
```


```
Average Dev Loss: 0.38185237123618065
Average Dev accuracy: 0.0
Average Dev F1: 0.8988944260272429
Average train Loss: 0.2163824252063645
Current Learning Rate: 0.01
```


```
Best sklearn masked F1: 0.8988944260272429
Epoch: 6
100%|██████████| 1874/1874 [00:18<00:00, 101.60it/s]
100%|██████████| 434/434 [00:02<00:00, 176.17it/s]
Average Dev Loss: 0.3418062004123381
Average Dev accuracy: 0.0
Average Dev F1: 0.9031941570597836
Average train Loss: 0.17325236418074755
Current Learning Rate: 0.01
Best sklearn masked F1: 0.9031941570597836
Epoch: 7
100%|██████████| 1874/1874 [00:18<00:00, 103.73it/s]
100%|██████████| 434/434 [00:02<00:00, 179.76it/s]
Average Dev Loss: 0.40430830066598744
Average Dev accuracy: 0.0
Average Dev F1: 0.9451792849373514
Average train Loss: 0.1486453927482857
Current Learning Rate: 0.01
Best sklearn masked F1: 0.9451792849373514
Epoch: 8
100%|██████████| 1874/1874 [00:18<00:00, 101.19it/s]
100%|██████████| 434/434 [00:02<00:00, 179.87it/s]
Average Dev Loss: 0.42299528956876786
Average Dev accuracy: 0.0
Average Dev F1: 0.9401154687470477
Average train Loss: 0.12461662309203524
Current Learning Rate: 0.01
Best sklearn masked F1: 0.9451792849373514
Epoch: 9
100%|██████████| 1874/1874 [00:18<00:00, 100.40it/s]
100%|██████████| 434/434 [00:02<00:00, 176.38it/s]
Average Dev Loss: 0.3905213369605934
Average Dev accuracy: 0.0
Average Dev F1: 0.9132165193150874
Average train Loss: 0.1220061460059144
Current Learning Rate: 0.01
Best sklearn masked F1: 0.9451792849373514
Epoch: 10
100%|██████████| 1874/1874 [00:18<00:00, 104.06it/s]
100%|██████████| 434/434 [00:02<00:00, 176.00it/s]
Average Dev Loss: 0.36011768190494914
Average Dev accuracy: 0.0
Average Dev F1: 0.9140407793385151
Epoch 00011: reducing learning rate of group 0 to 5.5000e-03.
Average train Loss: 0.13264057458837575
Current Learning Rate: 0.0055000000000000005
Best sklearn masked F1: 0.9451792849373514
Epoch: 11
```

```
100%|██████████| 1874/1874 [00:18<00:00, 103.28it/s]
100%|██████████| 434/434 [00:02<00:00, 158.76it/s]
Average Dev Loss: 0.40372101074054595
Average Dev accuracy: 0.0
Average Dev F1: 0.9632844905972828
Average train Loss: 0.027640321590413866
Current Learning Rate: 0.0005
Best sklearn masked F1: 0.9667740115298565
Epoch: 99
100%|██████████| 1874/1874 [00:18<00:00, 101.40it/s]
100%|██████████| 434/434 [00:02<00:00, 164.83it/s]
Average Dev Loss: 0.4062223510756608
Average Dev accuracy: 0.0
Average Dev F1: 0.9620557984641228
Average train Loss: 0.02718517045453668
Current Learning Rate: 0.0005
Best sklearn masked F1: 0.9667740115298565
```

Notes:

- Embedding initialization helps a lot
- Model learns faster
- Dev loss is less compared to the vanilla LSTM which supports our intuition

```
!perl conll03eval < dev2.out
```

```
processed 51577 tokens with 5942 phrases; found: 6190 phrases; correct: 5066.
accuracy:  96.98%; precision:  81.84%; recall:  85.26%; FB1:  83.51
              LOC: precision:  88.91%; recall:  92.54%; FB1:  90.69  1912
             MISC: precision:  70.30%; recall:  73.43%; FB1:  71.83  963
              ORG: precision:  78.05%; recall:  79.27%; FB1:  78.65  1362
              PER: precision:  83.26%; recall:  88.27%; FB1:  85.69  1953
```

+ Code + Markdown

Remark: Better hyperparameter is needed to increase the performance.