TASK 1

In [1]:

```
import numpy as np
import torch
import torch.utils.data as data_utils
import torch.nn as nn
import torch.nn.functional as F
import copy
from sklearn.metrics import classification_report
import warnings
import task1 as t1
```

```
/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:138: UserWarning:
A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (de
tected version 1.24.1)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is re
quired for this version of "
```

In [2]:

```
warnings.filterwarnings('ignore')
```

In [3]:

```
# read all data into list of lists
train_list = t1.read_file_to_list('data/train')
dev_list = t1.read_file_to_list('data/dev')
test_list = t1.read_file_to_list('data/test')
```

In [4]:

```
# create corpus from training data / index mapping
dict_word2idx = t1.create_corpus(train_list)
dict_ner2idx = {'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4,
                'B-LOC': 5, 'I-LOC': 6, 'B-MISC': 7, 'I-MISC': 8}
dict_ind2ner = {v: k for k, v in dict_ner2idx.items()}
```

In [5]:

```
# prepare data (map with index, padding)
X_train, y_train, mask_train = t1.prepare_data_mask(train_list, dict_word2idx
X_dev, y_dev, mask_dev = t1.prepare_data_mask(dev_list, dict_word2idx, dict_n
X_test, mask_test = t1.prepare_data_mask(test_list, dict_word2idx, dict_ner2i
```

In [6]:

```
# calculate weights of classes to be used in nn.CrossEntropyLoss
weights = dict()
for sent in y_train:
    for lab in sent:
        if lab != -1:
            weights[lab] = weights.get(lab, 0) + 1
weights = [i for _, i in sorted(weights.items())]
weights = torch.tensor(weights) / sum(weights)
weights = 1. / weights
# weights[0] = 0.01
# weights = torch.tensor([0.8, 1, 1, 0.9, 1, 1, 1, 1, 1])
```

In [7]:

```
# X_train, y_train = X_train[:100], y_train[:100]
# X_dev, y_dev = X_dev[:100], y_dev[:100]
# mask_train = mask_train[:100]
# mask_dev = mask_dev[:100]
```

In [8]:
```
# convert data to tensors & dataloaders
X_train_tensor = torch.from_numpy(X_train)
y_train_tensor = torch.from_numpy(y_train)
mask_train_tensor = torch.from_numpy(mask_train)
train_tensor = data_utils.TensorDataset(X_train_tensor, mask_train_tensor, y_
train_loader = data_utils.DataLoader(train_tensor, batch_size=10, shuffle=Tru

X_dev_tensor = torch.from_numpy(X_dev)
y_dev_tensor = torch.from_numpy(y_dev)
mask_dev_tensor = torch.from_numpy(mask_dev)
dev_tensor = data_utils.TensorDataset(X_dev_tensor, mask_dev_tensor, y_dev_te
dev_loader = data_utils.DataLoader(dev_tensor, batch_size=10, shuffle=True)

X_test_tensor = torch.from_numpy(X_test)
mask_test_tensor = torch.from_numpy(mask_test)
```

In [9]:
```
# initialize the NN
model = t1.model_BiLSTM(len(dict_word2idx), len(dict_ner2idx))
print(model)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss(ignore_index=-1, reduction='mean')

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
```

```
model_BiLSTM(
  (embedding): Embedding(23624, 100)
  (lstm): LSTM(101, 256, batch_first=True, dropout=0.33, bidirectional=True)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

In [10]:
```
best_sd = t1.train(20, model, train_loader, dev_loader, optimizer, criterion,
                X_train_tensor, y_train_tensor, X_dev_tensor, y_dev_tensor, m
                mask_dev_tensor, print_every=1)
```

```
Epoch: 1        Train Loss: 0.026 || Valid Loss: 0.015 || Train Acc: 0.937 ||
Valid Acc: 0.932 || Valid F1: 0.607
Epoch: 2        Train Loss: 0.013 || Valid Loss: 0.013 || Train Acc: 0.964 ||
Valid Acc: 0.943 || Valid F1: 0.683
Epoch: 3        Train Loss: 0.008 || Valid Loss: 0.012 || Train Acc: 0.978 ||
Valid Acc: 0.951 || Valid F1: 0.713
Epoch: 4        Train Loss: 0.005 || Valid Loss: 0.013 || Train Acc: 0.988 ||
Valid Acc: 0.951 || Valid F1: 0.732
Epoch: 5        Train Loss: 0.003 || Valid Loss: 0.018 || Train Acc: 0.993 ||
Valid Acc: 0.947 || Valid F1: 0.728
Epoch: 6        Train Loss: 0.002 || Valid Loss: 0.016 || Train Acc: 0.997 ||
Valid Acc: 0.953 || Valid F1: 0.748
Epoch: 7        Train Loss: 0.001 || Valid Loss: 0.019 || Train Acc: 0.998 ||
Valid Acc: 0.952 || Valid F1: 0.746
Epoch: 8        Train Loss: 0.000 || Valid Loss: 0.015 || Train Acc: 0.999 ||
Valid Acc: 0.958 || Valid F1: 0.760
Epoch: 9        Train Loss: 0.000 || Valid Loss: 0.018 || Train Acc: 1.000 ||
Valid Acc: 0.956 || Valid F1: 0.761
Epoch: 10       Train Loss: 0.000 || Valid Loss: 0.018 || Train Acc: 1.000 ||
Valid Acc: 0.957 || Valid F1: 0.764
Epoch: 11       Train Loss: 0.000 || Valid Loss: 0.019 || Train Acc: 1.000 ||
Valid Acc: 0.956 || Valid F1: 0.765
Epoch: 12       Train Loss: 0.000 || Valid Loss: 0.018 || Train Acc: 1.000 ||
Valid Acc: 0.958 || Valid F1: 0.764
Epoch: 13       Train Loss: 0.000 || Valid Loss: 0.020 || Train Acc: 1.000 ||
```

```
Valid Acc: 0.956 || Valid F1: 0.764
Epoch: 14       Train Loss: 0.000 || Valid Loss: 0.020 || Train Acc: 1.000 ||
Valid Acc: 0.957 || Valid F1: 0.765
Epoch: 15       Train Loss: 0.000 || Valid Loss: 0.020 || Train Acc: 1.000 ||
Valid Acc: 0.957 || Valid F1: 0.765
Epoch: 16       Train Loss: 0.000 || Valid Loss: 0.021 || Train Acc: 1.000 ||
Valid Acc: 0.956 || Valid F1: 0.763
Epoch: 17       Train Loss: 0.000 || Valid Loss: 0.020 || Train Acc: 1.000 ||
Valid Acc: 0.957 || Valid F1: 0.763
Epoch: 18       Train Loss: 0.000 || Valid Loss: 0.021 || Train Acc: 1.000 ||
Valid Acc: 0.956 || Valid F1: 0.765
Epoch: 19       Train Loss: 0.000 || Valid Loss: 0.021 || Train Acc: 1.000 ||
Valid Acc: 0.957 || Valid F1: 0.762
Epoch: 20       Train Loss: 0.000 || Valid Loss: 0.021 || Train Acc: 1.000 ||
Valid Acc: 0.957 || Valid F1: 0.765
```

In [11]:

```
model_best = t1.model_BiLSTM(len(dict_word2idx), len(dict_ner2idx))
model_best.load_state_dict(best_sd)

train_acc, train_prec, train_rec, train_f1, _ = t1.report_scores(model_best(X
dev_acc, dev_prec, dev_rec, dev_f1, _ = t1.report_scores(model_best(X_dev_ten

print('Training:')
print('Accuracy = {:.4f}, Precision = {:.4f}, Recall = {:.4f}, F1-score = {:.
    train_acc, train_prec, train_rec, train_f1))
print('Testing:')
print('Accuracy = {:.4f}, Precision = {:.4f}, Recall = {:.4f}, F1-score = {:.
    dev_acc, dev_prec, dev_rec, dev_f1))
```

```
Training:
Accuracy = 0.9999, Precision = 0.9995, Recall = 0.8561, F1-score = 0.9223
Testing:
Accuracy = 0.9572, Precision = 0.8546, Recall = 0.6931, F1-score = 0.7654
```

In [12]:

```
# get predictions
y_pred_train = torch.argmax(model_best(X_train_tensor, mask_train_tensor), di
y_pred_dev = torch.argmax(model_best(X_dev_tensor, mask_dev_tensor), dim=-1)
y_pred_test = torch.argmax(model_best(X_test_tensor, mask_test_tensor), dim=-

y_pred_train_format = t1.format_prediction(train_list, y_pred_train, dict_ind
y_pred_dev_format = t1.format_prediction(dev_list, y_pred_dev, dict_ind2ner)
y_pred_test_format = t1.format_prediction(test_list, y_pred_test, dict_ind2ne

merge_pred_train = t1.merge_truth_pred(train_list, y_pred_train_format)
merge_pred = t1.merge_truth_pred(dev_list, y_pred_dev_format)
```

In [13]:

```
# write files
t1.write_file(y_pred_dev_format, 'dev1.out')
t1.write_file(y_pred_test_format, 'test1.out')
t1.write_file(merge_pred, 'merge_dev_1.txt')
t1.write_file(merge_pred_train, 'merge_train_1.txt')
```

In [14]:

```
# save models
t1.save_model(model_best, 'blstm1.pt')
```

In [16]:

```
y_pred = model_best(X_dev_tensor, mask_dev_tensor)
y_true = y_dev_tensor

prediction = torch.argmax(y_pred, dim=-1)
mask = y_true > -1
```

```
prediction = prediction[mask]

# accuracy
num_match = (prediction == y_true[mask]).sum()
num_total = prediction.shape[0]
acc = (num_match / num_total).item()

# precision, recall, f1
dict_report = classification_report(y_true[mask], prediction)
print(dict_report)
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99     41273
           1       0.90      0.70      0.79      1737
           2       0.93      0.80      0.86      1199
           3       0.47      0.90      0.62      1317
           4       0.83      0.76      0.80       727
           5       0.92      0.84      0.88      1778
           6       0.85      0.81      0.83       244
           7       0.93      0.77      0.84       901
           8       0.89      0.71      0.79       331

    accuracy                           0.96     49507
   macro avg       0.86      0.81      0.82     49507
weighted avg       0.97      0.96      0.96     49507
```

TASK 1 Explanation

> - What are the precision, recall and F1 score on the dev data?
>   Precision = 79.57%
>   Recall = 79.33%
>   F1 score = 79.45%
>
> - Description of the solution
>   In the data cleaning/preprocessing step, the corpus is created from training data to collect the index of different words. Then, the data (sentences) is converted to a list of index. Furthermore, the length of the sentences is truncated to the maximum length of 35 to reduce the training time, and the sentence with shorter lengths are padded. Then, the prepared data is passed to the first layer of the NN module which is the embedding layer. CrossEntropyLoss is used along with SGD as an optimizer. The weights (which are the inverse of the class frequency) are not used in this task. Moreover, 4 different masks were used to further improve the model's performance: a mask that indicates wheter a word is capitalized, as mask that indicates whether a word contains a number, a mask that indicates if the word is the first one in the sentence, and a mask that indicates whether the word is all uppercase.
>
> - Hyper-parameters used in the network architecture
>   batch size used is 10, learning rate used is 0.5, and no learning rate scheduler is used
>   the rest of the architecture is as per instruction, where first layer is the embedding layer that generates the output with the dimension of 100,

> then the Bidirectional LSTM layers with hidden dimension of 256 and dropout rate of 0.33, and followed by a linear unit, an ELU, and a classifier to classify NER to 9 different classes.

References

- https://gangaksankar.medium.com/lstm-model-for-ner-tagging-7c2018c51ece

TASK 2

In [1]:
```
import numpy as np
import torch
import torch.utils.data as data_utils
import torch.nn as nn
import torch.nn.functional as F
import copy
from sklearn.metrics import classification_report
import gzip
import task2 as t2
import warnings
```

```
/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:138: UserWarning:
A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (de
tected version 1.24.1)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is re
quired for this version of "
```

In [2]:
```
warnings.filterwarnings('ignore')
```

In [3]:
```
# read all data into list of lists
train_list = t2.read_file_to_list('data/train')
dev_list = t2.read_file_to_list('data/dev')
test_list = t2.read_file_to_list('data/test')
```

In [4]:
```
# get GloVe embeddings / index mapping
glove = t2.get_embeddings('glove.6B.100d.gz')
dict_ner2idx = {'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4,
                'B-LOC': 5, 'I-LOC': 6, 'B-MISC': 7, 'I-MISC': 8}
dict_ind2ner = {v: k for k, v in dict_ner2idx.items()}
```

In [5]:
```
# prepare data (map with index, padding)
X_train, y_train = t2.prepare_data(train_list, glove, dict_ner2idx)
X_dev, y_dev = t2.prepare_data(dev_list, glove, dict_ner2idx)
X_test = t2.prepare_data(test_list, glove, dict_ner2idx, true_label=False)
```

In [6]:
```
# calculate weights of classes to be used in nn.CrossEntropyLoss
weights = dict()
for sent in y_train:
    for lab in sent:
        if lab != -1:
            weights[lab] = weights.get(lab, 0) + 1
weights = [i for _, i in sorted(weights.items())]
weights = torch.tensor(weights) / sum(weights)
weights = 1. / weights
```

In [7]:
```
# to run sample code
# X_train, y_train = X_train[:100], y_train[:100]
# X_dev, y_dev = X_dev[:100], y_dev[:100]
```

In [8]:

```
# convert data to tensors & dataloaders
X_train_tensor = torch.from_numpy(X_train.astype(np.float32))
y_train_tensor = torch.from_numpy(y_train)
train_tensor = data_utils.TensorDataset(X_train_tensor, y_train_tensor)
train_loader = data_utils.DataLoader(train_tensor, batch_size=10, shuffle=Tru

X_dev_tensor = torch.from_numpy(X_dev.astype(np.float32))
y_dev_tensor = torch.from_numpy(y_dev)
dev_tensor = data_utils.TensorDataset(X_dev_tensor, y_dev_tensor)
dev_loader = data_utils.DataLoader(dev_tensor, batch_size=10, shuffle=True)

X_test_tensor = torch.from_numpy(X_test.astype(np.float32))
```

In [9]:

```
# initialize the NN
model = t2.model_BiLSTM(len(dict_ner2idx))
print(model)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss(ignore_index=-1, weight=weights, reduction='m

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
```

```
model_BiLSTM(
  (lstm): LSTM(101, 256, batch_first=True, dropout=0.33, bidirectional=True)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier1): Linear(in_features=128, out_features=32, bias=True)
  (classifier2): Linear(in_features=32, out_features=9, bias=True)
)
```

In [10]:

```
best_sd = t2.train(20, model, train_loader, dev_loader, optimizer, criterion,
                    X_train_tensor, y_train_tensor, X_dev_tensor, y_dev_tenso
```

```
Epoch: 1        Train Loss: 0.062 || Valid Loss: 0.036 || Train Acc: 0.939 ||
Valid Acc: 0.941 || Valid F1: 0.726
Epoch: 2        Train Loss: 0.033 || Valid Loss: 0.027 || Train Acc: 0.940 ||
Valid Acc: 0.938 || Valid F1: 0.724
Epoch: 3        Train Loss: 0.025 || Valid Loss: 0.027 || Train Acc: 0.806 ||
Valid Acc: 0.801 || Valid F1: 0.621
Epoch: 4        Train Loss: 0.018 || Valid Loss: 0.021 || Train Acc: 0.943 ||
Valid Acc: 0.940 || Valid F1: 0.747
Epoch: 5        Train Loss: 0.015 || Valid Loss: 0.020 || Train Acc: 0.963 ||
Valid Acc: 0.959 || Valid F1: 0.785
Epoch: 6        Train Loss: 0.012 || Valid Loss: 0.022 || Train Acc: 0.928 ||
Valid Acc: 0.921 || Valid F1: 0.735
Epoch: 7        Train Loss: 0.010 || Valid Loss: 0.021 || Train Acc: 0.948 ||
Valid Acc: 0.944 || Valid F1: 0.762
Epoch: 8        Train Loss: 0.011 || Valid Loss: 0.023 || Train Acc: 0.968 ||
Valid Acc: 0.959 || Valid F1: 0.776
Epoch: 9        Train Loss: 0.007 || Valid Loss: 0.027 || Train Acc: 0.977 ||
Valid Acc: 0.970 || Valid F1: 0.813
Epoch: 10       Train Loss: 0.005 || Valid Loss: 0.024 || Train Acc: 0.980 ||
Valid Acc: 0.970 || Valid F1: 0.813
Epoch: 11       Train Loss: 0.005 || Valid Loss: 0.026 || Train Acc: 0.974 ||
Valid Acc: 0.965 || Valid F1: 0.792
Epoch: 12       Train Loss: 0.004 || Valid Loss: 0.034 || Train Acc: 0.981 ||
Valid Acc: 0.970 || Valid F1: 0.813
Epoch: 13       Train Loss: 0.003 || Valid Loss: 0.035 || Train Acc: 0.980 ||
Valid Acc: 0.969 || Valid F1: 0.811
Epoch: 14       Train Loss: 0.002 || Valid Loss: 0.033 || Train Acc: 0.976 ||
Valid Acc: 0.965 || Valid F1: 0.804
Epoch: 15       Train Loss: 0.002 || Valid Loss: 0.031 || Train Acc: 0.986 ||
Valid Acc: 0.974 || Valid F1: 0.823
```

```
Epoch: 16       Train Loss: 0.002 || Valid Loss: 0.035 || Train Acc: 0.983 ||
Valid Acc: 0.972 || Valid F1: 0.821
Epoch: 17       Train Loss: 0.001 || Valid Loss: 0.036 || Train Acc: 0.985 ||
Valid Acc: 0.973 || Valid F1: 0.825
Epoch: 18       Train Loss: 0.001 || Valid Loss: 0.039 || Train Acc: 0.993 ||
Valid Acc: 0.980 || Valid F1: 0.839
Epoch: 19       Train Loss: 0.001 || Valid Loss: 0.045 || Train Acc: 0.994 ||
Valid Acc: 0.981 || Valid F1: 0.840
Epoch: 20       Train Loss: 0.001 || Valid Loss: 0.044 || Train Acc: 0.989 ||
Valid Acc: 0.976 || Valid F1: 0.826
Epoch: 21       Train Loss: 0.003 || Valid Loss: 0.044 || Train Acc: 0.987 ||
Valid Acc: 0.975 || Valid F1: 0.816
Epoch: 22       Train Loss: 0.004 || Valid Loss: 0.033 || Train Acc: 0.984 ||
Valid Acc: 0.970 || Valid F1: 0.808
Epoch: 23       Train Loss: 0.003 || Valid Loss: 0.036 || Train Acc: 0.978 ||
Valid Acc: 0.966 || Valid F1: 0.801
Epoch: 24       Train Loss: 0.004 || Valid Loss: 0.046 || Train Acc: 0.987 ||
Valid Acc: 0.974 || Valid F1: 0.822
Epoch: 25       Train Loss: 0.003 || Valid Loss: 0.040 || Train Acc: 0.990 ||
Valid Acc: 0.977 || Valid F1: 0.825
Epoch: 26       Train Loss: 0.002 || Valid Loss: 0.039 || Train Acc: 0.988 ||
Valid Acc: 0.974 || Valid F1: 0.819
Epoch: 27       Train Loss: 0.001 || Valid Loss: 0.040 || Train Acc: 0.993 ||
Valid Acc: 0.980 || Valid F1: 0.838
Epoch: 28       Train Loss: 0.001 || Valid Loss: 0.049 || Train Acc: 0.994 ||
Valid Acc: 0.980 || Valid F1: 0.835
Epoch: 29       Train Loss: 0.002 || Valid Loss: 0.044 || Train Acc: 0.990 ||
Valid Acc: 0.977 || Valid F1: 0.823
Epoch: 30       Train Loss: 0.003 || Valid Loss: 0.034 || Train Acc: 0.985 ||
Valid Acc: 0.972 || Valid F1: 0.809
```

In [11]:

```
# report scores of best model
model_best = t2.model_BiLSTM(len(dict_ner2idx))
model_best.load_state_dict(best_sd)

train_acc, train_prec, train_rec, train_f1, _ = t2.report_scores(model_best(X
dev_acc, dev_prec, dev_rec, dev_f1, _ = t2.report_scores(model_best(X_dev_ten

print('Training:')
print('Accuracy = {:.4f}, Precision = {:.4f}, Recall = {:.4f}, F1-score = {:.
    train_acc, train_prec, train_rec, train_f1))
print('Testing:')
print('Accuracy = {:.4f}, Precision = {:.4f}, Recall = {:.4f}, F1-score = {:.
    dev_acc, dev_prec, dev_rec, dev_f1))
```

```
Training:
Accuracy = 0.9941, Precision = 0.9664, Recall = 0.8545, F1-score = 0.9070
Testing:
Accuracy = 0.9806, Precision = 0.9037, Recall = 0.7842, F1-score = 0.8397
```

In [12]:

```
# get predictions
y_pred_dev = torch.argmax(model_best(X_dev_tensor), dim=-1)
y_pred_test = torch.argmax(model_best(X_test_tensor), dim=-1)

y_pred_dev_format = t2.format_prediction(dev_list, y_pred_dev, dict_ind2ner)
y_pred_test_format = t2.format_prediction(test_list, y_pred_test, dict_ind2ne

merge_pred = t2.merge_truth_pred(dev_list, y_pred_dev_format)
```

In [13]:

```
# write files
t2.write_file(y_pred_dev_format, 'dev2_new2.out')
t2.write_file(y_pred_test_format, 'test2_new2.out')
t2.write_file(merge_pred, 'merge_dev_2_new2.txt')
```

In [14]:
```
# save models
t2.save_model(model_best, 'blstm2_new2.pt')
```

In [15]:
```
y_pred = model_best(X_dev_tensor)
y_true = y_dev_tensor

prediction = torch.argmax(y_pred, dim=-1)
mask = y_true > -1
prediction = prediction[mask]

# accuracy
num_match = (prediction == y_true[mask]).sum()
num_total = prediction.shape[0]
acc = (num_match / num_total).item()

# precision, recall, f1
dict_report = classification_report(y_true[mask], prediction)
print(dict_report)
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      0.99     41273
           1       0.94      0.96      0.95      1737
           2       0.97      0.96      0.97      1199
           3       0.86      0.92      0.89      1317
           4       0.85      0.87      0.86       727
           5       0.93      0.96      0.95      1778
           6       0.91      0.92      0.91       244
           7       0.88      0.88      0.88       901
           8       0.79      0.77      0.78       331

    accuracy                           0.98     49507
   macro avg       0.90      0.91      0.91     49507
weighted avg       0.98      0.98      0.98     49507
```

TASK 2 Explanation

> - What are the precision, recall and F1 score on the dev data? (using conll03eval script)
>   Precision = 88.03%
>   Recall = 88.88%
>   F1 score = 88.45%
>
> - Description of the solution
>   In the data cleaning/preprocessing step, the GloVe embeddings are used to represent each word in the sentences. Glove's case insensitivity issue is dealt with by taking lowercase of the input sentence. Furthermore, the length of the sentences is truncated to the maximum length of 35 to reduce the training time, and the sentence with shorter lengths are padded with vectors of 0's. Then, the prepared data is passed to the bidirectional LSTM network. CrossEntropyLoss is used along with SGD as an optimizer. The weights (which are the inverse of the class frequency) are passed to the CrossEntropyLoss too to deal with class imbalance.

Moreover, a mask that indicates whether a word is capitalized or not is added as another dimension to tackle the issue that GloVe does not consider capitalization.

- Hyper-parameters used in the network architecture
batch size used is 10, learning rate used is 0.5, and no learning rate scheduler is used
the rest of the architecture is as per instruction, where first layer is the embedding layer that generates the output with the dimension of 100, then the Bidirectional LSTM layers with hidden dimension of 256 and dropout rate of 0.33, and followed by a linear unit, an ELU, and a classifier to classify NER to 9 different classes.

References

- https://medium.com/analytics-vidhya/basics-of-using-pre-trained-glove-vectors-in-python-d38905f356db