CSCI544: Homework Assignment No4

Task 1 : Simple Bidirectional LSTM model

What are the precision, recall and F1 score on the dev data?

Using SkLearn:

Precision: 0.9496

Recall: 0.9518

F1 Score: 0.9507

Using official evaluation script:

Precision: 80.92%

Recall: 72.53%

F1 Score: 76.50

Defined hyper parameters for the model:

EMBEDDING_DIM = 100

HIDDEN_DIM = 256

OUTPUT_DIM = 128

DROPOUT = 0.33

LEARNING_RATE = 1

EPOCHS = 111

BATCH_SIZE = 64

This code implements a Named Entity Recognition (NER) model for identifying named entities in text.

1. Data preprocessing:

The code begins by importing necessary libraries and setting the device for running the code (Task 1 was run on GPU). It also defines some hyperparameters such as the dimensions of the embedding (100) and hidden layers dimension (256), dropout rate (0.33), learning rate (1), and batch size (64). The code then reads in training and development data and stores the data in separate lists for words and tags. It also creates separate lists that combine the words and tags for each sentence in the training and development data (train_set_words - list of sentences, train_set_tags - list of tags for each sentence). The code then builds a vocabulary of words (unique words) and a frequency dictionary of tags based on the training data. It also creates a mapping from each word and tag to a unique integer index. In word_vocab, three special tokens are being added: <pad>, which is used to pad sequences to a fixed length and is assigned a value of -1 in both the word and tag vocabularies; <unk>, which represents unknown words not in the vocabulary and is assigned a value of 0 in the word vocabulary; and <pad> in the tag vocabulary, which is also assigned a value of -1, indicating it should be ignored during training and evaluation since it is not a valid tag.

We later replaced the words with frequency less than 3 with '<unk>'. As words that occur infrequently in the training data may not provide enough information to the model to learn useful patterns and relationships, leading to overfitting or poor generalization performance on new data.

The code then defines a dataset class for loading the data, which includes methods for converting words and tags to integer vectors (using their respective indexes). It also defines a collate function for padding sentences and tags.

2. Bidirectional architecture:

The neural network model is a bidirectional LSTM followed by a linear layer, and it is trained using a cross-entropy loss function and SGD optimizer.

The BiLSTM class consists of an embedding layer that maps each word in the input sequence to a low-dimensional vector representation. These embeddings are then passed through a bidirectional LSTM layer, which processes the sequence both forwards and backwards, capturing contextual information from both directions.

Embedding layer -> BiLSTM (with dropout) -> Linear -> ELU -> Classifier

The forward method takes as input the input sequence x, initial hidden and cell states, sequence lengths, and returns the output of the model. The input sequence is first embedded using the embedding layer, and then packed using PyTorch's pack_padded_sequence function which allows variable length sequences to be processed in a batched manner. The packed sequence is then fed to the LSTM layer, along with the initial hidden and cell states. The output is then padded back to the original length using pad_packed_sequence. The output of the LSTM layer is passed through a dropout layer to prevent overfitting, and then through a fully connected layer with an ELU activation function. Finally, the output is passed through a linear layer with output dimension 9, which represents the number of possible tags. The initHidden and initCellStates methods initialize the hidden and cell states with zeros for a given batch size.

3. Training and testing:

During training, the negative log likelihood of the correct labels is minimized using the cross-entropy loss function. I initialized cross entropy with class weights. The training loop consists of iterating over the dataset for a certain number of epochs, where for each epoch, the training data is fed into the model and the loss is calculated based on the predicted output and the true label. The loss is then backpropagated to update the weights of the model. The accuracy and loss of the model are calculated and printed after each epoch. The code also includes a checkpoint mechanism to save the state of the model at each epoch and to keep track of the best accuracy achieved. Finally, a learning rate scheduler used to adjust the learning rate at each epoch.

The correct_dev variable keeps track of the number of correct predictions made by the model. It uses the np.count_nonzero() function to count the number of correct predictions for each sequence in the batch. The predicted_labels and true_labels lists store the predicted and true labels, respectively, for each sequence in the batch.

4. Metrics:

At the end of the evaluation loop, predicted_labels and true_labels contain the predicted and true labels for all sequences in the dev data. These lists can be used to evaluate the performance of the model using precision, recall, and F1-score from sklearn.

Task 2 : Using GloVe word embeddings

What are the precision, recall and F1 score on the dev data?

Using SkLearn:

Precision: 0.9691

Recall: 0.9701

F1 Score: 0.9696

Using official evaluation script:

Precision: 87.76%

Recall: 82.87%

F1 Score: 85.24

Defined hyper parameters for the model:

EMBEDDING_DIM = 100

HIDDEN_DIM = 256

OUTPUT_DIM = 128

DROPOUT = 0.33

LEARNING_RATE = 1

EPOCHS = 30

BATCH_SIZE = 64

I used the same code for task 2 with changes to embeddings and Bidirectional LSTM:

Added code that loads pre-trained word embeddings from a file called 'glove.6B.100d' and constructs a word-to-vector dictionary by parsing each line of the file. Then, it creates an embedding matrix, with the number of rows equal to the number of words in the vocabulary and the number of columns equal to the embedding dimension. For each word in the vocabulary, the code checks whether a corresponding word vector exists in the word-to-vector dictionary. If so, it adds the vector to the embedding matrix; otherwise, it assigns a random vector to that word. Finally, the code loops through each sentence in the training set and replaces words with a frequency of less than 3 with a special '<unk>' token, effectively treating them as unknown words during training.

BiLSTM architecture:

The input to the network is a sequence of tokens, which are first converted to their corresponding pre-trained GloVe embeddings and concatenated with a 5 dimensional vector using a binary indicator to mark whether the word is capitalized or not. The concatenated embeddings are then packed into a padded sequence using pack_padded_sequence() to eliminate the effects of padding on the network's computations. The packed sequence is passed through the bidirectional LSTM layer, which has a hidden state of size 256.

The output of the LSTM layer is then passed through a linear layer, followed by an ELU activation function, and finally a classifier layer to produce the final output of size 9. The dropout parameter controls the dropout rate for regularization. The class also includes the initHidden() and initCellStates() functions to initialize the hidden and cell states of the LSTM layer, respectively.

I have a "word embedding layer", which is of the shape (vocab_size, 100). and also created a "capitalization embedding layer", which is of the shape (2, 5). Given a word, the input to LSTM will be the concatenation of the embedding derived from the "word embedding layer", which is based on its lower-cased form, and an embedding derived from the "capitalization embedding layer", which is a short vector indicating whether the word is lower-cased or not. This is how I handled case-sensitive words.