CSCI 544 – Applied Natural Language Processing

Homework 4

Rishi Kiran Reddy Nareddy
ID: 1582852239

Prerequisites:

- Python version: 3.9.12
- PyTorch Version: 1.13.1
- Packages used:
  i) import torch
  ii) import torch.nn as nn
  iii) import torch.optim as optim
  iv) import classification_report
  v) import precision_score, recall_score, f1_score
  vi) import numpy as np

Task1 Simple Bidirectional LSTM Model:

1. **Reading Data:** We read the contents of the file named "train" using the built-in **open()** function in read mode ('r'). It then reads all the lines in the file using the **readlines()** method, and stores them as a list of strings in the lines variable.

   The for loop then iterates through each string in the lines list, using the **range()** function to generate an index for each string. For each string, the **split()** method is called to split the string into a list of words using whitespace as the delimiter. The resulting list of words is then assigned to the original position of the string in the lines list.

   In summary, we read a text file and splits each line into a list of words, which can be used for further processing such as training a model or performing data analysis.

2. **Data Preparation:** We initialize two empty lists named sentences and tags. It also initializes two empty lists named **current_sentence** and **current_tags**.

   The code then iterates through each line in the lines list. For each line, it checks whether its length is 0 or not. If the length is 0, it means that a new sentence has started. In this case, it appends the **current_sentence** and **current_tags** lists to the sentences and tags lists, respectively. It also re-initializes the **current_sentence** and **current_tags** lists to empty lists. Otherwise, it appends the second and third columns of the line (i.e., the word and its corresponding tag) to the **current_sentence** and **current_tags** lists, respectively.

   After iterating through all the lines, the sentences and tags lists will contain all the sentences and their corresponding tags in the train file, respectively.

   The code then initializes an empty dictionary named **word_to_ix**. It also initializes a dictionary named **tag_to_ix** that maps each possible named entity tag to a unique index.

Next, the code iterates through each sentence in the **sentences** list. For each sentence, it iterates through each word in the sentence and checks whether the word is already present in the **word_to_ix** dictionary or not. If the word is not present, it adds the word as a key in the **word_to_ix** dictionary with a value equal to the current length of the dictionary (i.e., a unique index is assigned to each unique word in the training data).

Finally, the sentences and tags lists are converted to PyTorch tensors, which can be fed into a PyTorch model for training. The **word_to_ix** and **tag_to_ix** dictionaries are also used to convert the words and tags to their corresponding indices during training.

3. **Network Architecture:** We define a PyTorch model called BLSTM which is a bidirectional LSTM model with two linear layers for named entity recognition (NER) task.

   The **__init__** function of the BLSTM class defines the structure of the model. It takes the following arguments:
   - **vocab_size:** an integer representing the size of the vocabulary (i.e., the number of unique words in the training data).
   - **tag_to_ix:** a dictionary that maps each possible named entity tag to a unique index.
   - **embedding_dim:** an integer representing the size of the word embedding.
   - **hidden_dim:** an integer representing the number of hidden units in the LSTM layer.
   - **dropout:** a float representing the dropout rate.

   The **__init__** function initializes the following layers:
   - **word_embeddings:** an nn.Embedding layer that maps each word to a dense vector of size embedding_dim.
   - **lstm:** an nn.LSTM layer that takes the word embeddings as input and outputs a sequence of hidden states.
   - **hidden1tag:** a linear layer that maps the hidden states to a lower-dimensional space.
   - **hidden2tag:** a linear layer that maps the lower-dimensional space to the output space, which has a size equal to the number of unique named entity tags.

   The forward function of the BLSTM class defines how the inputs are processed by the model. It takes a PyTorch tensor representing a sentence as input and returns a PyTorch tensor representing the predicted tags for each word in the sentence.

   The forward function first passes the sentence through the **word_embeddings** layer to get the word embeddings. It then passes the embeddings through the **lstm** layer to get the hidden states. The hidden states are then passed through the **hidden1tag** layer, the activation function (in this case, the **ELU** activation function), and the **hidden2tag** layer to get the predicted tags. The final output is a PyTorch tensor representing the predicted tags for each word in the sentence.

4. **Network Training:** We write a code that trains a Bidirectional LSTM (BLSTM) model for Named Entity Recognition (NER) on the given training data. Here is a step-by-step explanation of the code:
   - Hyperparameters are defined for the model. These include the embedding dimension (=100), hidden dimension (=256), output dimension (=128), dropout rate (=0.33), learning rate (=0.1), and number of epochs (=20).

- An instance of the BLSTM model is created with the defined hyperparameters. The model takes in the **vocabulary size, tag_to_ix dictionary, embedding dimension, hidden dimension, and dropout rate** as inputs.

- A loss function and optimizer are defined. The **CrossEntropyLoss** function is used as the loss function, and **Stochastic Gradient Descent (SGD)** is used as the optimizer. The model parameters are passed to the optimizer for optimization.

- The model is trained for the specified number of epochs. For each epoch, the model is trained on each sentence in the training data. For each sentence, the accumulated gradients are cleared, the forward pass is calculated, the loss is calculated, and backpropagation is performed to update the model parameters. The loss for the current epoch is printed after each epoch.

- The trained model can be used to predict tags for new sentences using the model1 object.

- Then we save the model using the **torch.save(model1, 'blstm1.pt')**.

    - **torch.save()** is a function in PyTorch that is used to save a model to disk. The function takes two arguments: the first argument is the model instance that we want to save, and the second argument is the file name and path where we want to save the model.
    - In the given code, **torch.save(model1,'blstm1.out')** saves the trained **model1** to disk with the file name **blstm1.out**. The saved file will contain the model's parameters, which can be loaded later using the **torch.load()** function. Saving the trained model is important because it allows us to use the model later without having to retrain it from scratch.

5. Model Evaluation:

- Then we write a code that is used for evaluating the performance of a trained BLSTM model on a separate development dataset. The development dataset is converted into a list of sentences and then each sentence is converted to a PyTorch tensor using the same word-to-index mapping used during the training phase.

- Next, the PyTorch tensor representing each sentence is passed through the trained BLSTM model to get the predicted tag scores for each word in the sentence. The predicted tag scores are stored in a list.

- Then, the predicted tags for each sentence are obtained by taking the argmax of the predicted tag scores along the dimension of the tags. The predicted tags are converted to their corresponding tag labels using the **tag_to_ix** dictionary.

- Finally, the predicted tags for each sentence are stored in a list called **dev_predicted_tags**. This list can be used to compare the performance of the model on the development dataset with its performance on the training dataset, or to tune hyperparameters such as the learning rate or number of epochs.

- Now, the **dev_predicted_tags** have to be flattened to be able to generate the **dev1.out** file. The dev1.out file is updated to be in a particular format for evaluating against the **conll03eval** script.

- Now, after the flattening the predicted tags, the precision, recall, and f1-score using the scikit learn with **import precision_score, recall_score, f1_score** is as follows:

```
Precision: 0.906, Recall: 0.731, F1 score: 0.806
```

- The result after executing the perl command "**perl conll03eval.txt < dev1.out**" is as follows:

```
(base) rishis-MBP:~ rishinareddy$ perl conll03eval.txt < dev1.out
processed 51578 tokens with 5942 phrases; found: 4961 phrases; correct: 4114.
accuracy:  94.82%; precision:  82.93%; recall:  69.24%; FB1:  75.47
              LOC: precision:  91.44%; recall:  77.95%; FB1:  84.16  1566
             MISC: precision:  84.46%; recall:  73.10%; FB1:  78.37  798
              ORG: precision:  76.08%; recall:  62.86%; FB1:  68.84  1108
              PER: precision:  78.24%; recall:  63.25%; FB1:  69.95  1489
```

6. **Test File generation:** We predict the named-entity-recognition tags for a test set using a trained model and write the predicted tags to an output file. The steps involved are:

    - Reading in the test data from a file named 'test' and splitting it into a list of lists where each inner list represents a sentence and contains the words and their corresponding tags.

    - Converting each sentence in the test set to a PyTorch tensor of word indices, similar to what was done for the development set in the previous code example.

    - Passing each sentence tensor through the trained model to obtain the predicted tag scores.

    - For each sentence, selecting the tag with the highest score as the predicted tag, and storing these predicted tags in a list.

    - Flattening the list of lists of predicted tags into a single list.

    - Writing the original test data, along with the predicted tags, to an output file named **'test1.out'**. The predicted tag for each word in a sentence is written on a separate line, along with the corresponding sentence index and word itself.

    - Overall, the code we write allows us to evaluate the performance of the trained model on an unseen test set and generate predictions for future use.

Task 2 Using Glove Embeddings:

We already read the training data in the task 1, So we can use that. In task 2, we also read the Glove Embeddings. In Task 1, the embeddings were learned from scratch during training using **nn.Embedding**. However, in Task 2, pre-trained GloVe embeddings were used instead, which were loaded from a file and used to initialize the embedding layer.
Here is how we read the Glove Embeddings:

- We read the pre-trained GloVe embeddings file, **glove.6B.100d**, and creates a dictionary mapping each word to its corresponding vector representation.

- The file glove.6B.100d contains 100-dimensional vectors for 400,000 words, trained on a corpus of 6 billion tokens.

- The code iterates over each line in the file and splits it into two parts: the first part contains the word, and the remaining parts contain the corresponding vector components. It then creates a **numpy** array from the vector components and adds the word and its vector representation as a key-value pair in the **word_to_vec** dictionary.

- The resulting word_to_vec dictionary can then be used to look up the vector representation for each word in a given sentence, which can be useful in tasks such as natural language processing and text classification.

To handle the conflict between GloVe's case-insensitivity, I check if the lowercase version of a word is in the pre-trained GloVe embeddings. If it is, the corresponding GloVe embedding is used. If not, a random embedding is generated. This approach allows the model to use GloVe embeddings while capturing capitalization information. Treating all words with the same lowercase form as equivalent ensures that the model is not biased towards capitalization in the pre-trained embeddings.

1) **Network Architecture:** The GloveBLSTM class is a PyTorch module that defines a Bidirectional LSTM (BLSTM) model for sequence tagging. This model takes as input a sentence of words, and produces as output a sequence of predicted tags for each word in the sentence.

   The model architecture consists of the following layers:

   - **Embedding layer:** This layer maps each word in the input sentence to a dense vector representation of size **embedding_dim**. The embedding layer is initialized with pre-trained GloVe word embeddings that are passed to the model as a parameter **word_embeddings**.

   - **Bidirectional LSTM layer:** This layer processes the sequence of embedded words in both forward and backward directions. The output of this layer is a sequence of hidden states, where each hidden state is a concatenation of the forward and backward states. The LSTM layer has a hidden size of **hidden_dim//2**, as it is bidirectional.

   - **Linear layer:** This layer applies a linear transformation to each hidden state to produce a new sequence of hidden states with size **hidden_dim//2**.

- **Dropout layer:** This layer randomly sets a fraction of the elements in the input tensor to zero during training, in order to prevent overfitting.

- **Activation layer:** This layer applies the **Exponential Linear Unit (ELU)** activation function to each element in the output of the linear layer.

- **Linear layer:** This layer applies a linear transformation to each element in the output of the activation layer to produce a sequence of predicted tag scores for each word in the input sentence. The output size of this layer is **tag_size**, which is the number of possible tags in the sequence tagging task.

The forward method of the GloveBLSTM class implements the forward pass of the model. It takes as input a sentence of words, and returns a PyTorch tensor of predicted tag scores for each word in the sentence.

The forward method first converts each word in the sentence to its corresponding GloVe embedding, or a randomly initialized embedding if the word is not in the pre-trained embeddings. The embeddings are concatenated into a tensor of shape (1, sequence_length, embedding_dim), where sequence_length is the length of the input sentence.

This tensor is then passed through the LSTM layer, which produces a sequence of hidden states of shape **(1, sequence_length, hidden_dim)**. The hidden states are then passed through the linear, dropout, activation, and final linear layers to produce the predicted tag scores for each word in the input sentence, which is a tensor of shape **(sequence_length, tag_size)**.

2) **Model Training:** The network architecture is a Bidirectional LSTM (BLSTM) with GloVe embeddings. It consists of an embedding layer, a BLSTM layer, two fully connected (linear) layers, a dropout layer, and an activation function.

   The GloveBLSTM class is defined as a subclass of nn.Module. It takes several parameters such as **vocab_size, tag_size, embedding_dim, hidden_dim, dropout, and word_embeddings**.

   The forward method takes a sentence as input and returns predicted tag scores for each word in the sentence. It first creates the GloVe embedding for each word in the sentence using the **word_embeddings** dictionary. If the word is not present in the dictionary, it randomly initializes the embedding. The embeddings are then passed through the BLSTM layer to get the output. The output is then passed through two fully connected layers with an activation function and dropout in between. Finally, the output of the second linear layer gives the tag scores for each word.

   The code then initializes the model, **loss function** (**CrossEntropyLoss**), and **optimizer (SGD)** with a **learning rate of 0.1**. It trains the model for 20 epochs by iterating over each sentence and its corresponding tags. The optimizer updates the parameters of the model using backpropagation with respect to the loss calculated for each sentence. The total loss for each epoch is calculated as the sum of losses over all the sentences.

The purpose of the code is to train a neural network model for named entity recognition task using GloVe embeddings and a BLSTM architecture, and output the loss for each epoch.

Then we save the model using the **torch.save(model2, 'blstm2.pt').**

- torch.save() is a function in PyTorch that is used to save a model to disk. The function takes two arguments: the first argument is the model instance that we want to save, and the second argument is the file name and path where we want to save the model.
- In the given code, **torch.save(model2,'blstm2.out')** saves the trained model1 to disk with the file name blstm1.out. The saved file will contain the model's parameters, which can be loaded later using the torch.load() function. Saving the trained model is important because it allows us to use the model later without having to retrain it from scratch.

3) **Model Evaluation:** We have 4 steps in evaluating model. They are as follows:
Step 1: Convert the **dev_lines** data to a list of **sentences**. In this step, each line in the **dev_lines** data is checked. If the line is empty, the **current_sentence** is appended to **dev_sentences** list, and a new **current_sentence** is initialized. Otherwise, the second item (i.e., word) in the line is appended to the **current_sentence**.

Step 2: Convert the **dev_sentences** to PyTorch tensors.
In this step, each sentence in **dev_sentences** is iterated over. For each word in the sentence, its corresponding index is retrieved from the **word_to_ix** dictionary. If the word is not present in the dictionary, then the index of the **<UNK>** token is used. The sentence indices are then converted to a PyTorch tensor and appended to **dev_X** list.

Step 3: Pass each dev sentence tensor through the model to get the predicted tag scores.
The model is put into evaluation mode using model2.eval(). Then, each sentence in **dev_sentences** is iterated over. For each sentence, the model2 is called with the sentence as input to get the predicted tag scores. The predicted tag scores for each sentence are stored in dev_tag_scores list.

Step 4: Get the predicted tags for each sentence using the **tag_to_ix** dictionary.
For each set of predicted tag scores in **dev_tag_scores**, the index with the highest score is identified using **torch.max(tag_scores, dim=1)**. Then, the index is used to retrieve the corresponding tag from the **tag_to_ix** dictionary. The predicted tags for each sentence are stored in **dev_predicted_tags** list.

Finally, the **dev_predicted_tags** list can be used to evaluate the performance of the model using appropriate evaluation metrics such as accuracy, precision, recall, and F1-score.

Now, the **dev_predicted_tags** have to be flattened to be able to generate the **dev2.out** file. The dev2.out file is updated to be in a particular format for evaluating against the conll03eval script.

Now, after the flattening the predicted tags, the precision, recall, and f1-score using the scikit learn with **import precision_score, recall_score, f1_score** is as follows:

```
Precision: 0.911, Recall: 0.835, F1 score: 0.870
```

The result after executing the perl command "**perl conll03eval.txt < dev2.out**" is as follows:

```
(base) rishis-MBP:~ rishinareddy$ perl conll03eval.txt < dev2.out
processed 51578 tokens with 5942 phrases; found: 5806 phrases; correct: 5155.
accuracy:  97.38%; precision:  88.79%; recall:  86.76%; FB1:  87.76
              LOC: precision:  92.53%; recall:  92.32%; FB1:  92.43  1833
             MISC: precision:  86.16%; recall:  77.66%; FB1:  81.69  831
              ORG: precision:  80.30%; recall:  75.99%; FB1:  78.08  1269
              PER: precision:  92.04%; recall:  93.59%; FB1:  92.81  1873
```

4) Test File generation: We predict the named-entity-recognition tags for a test set using a trained model and write the predicted tags to an output file. The steps involved are:

- Reading in the test data from a file named 'test' and splitting it into a list of lists where each inner list represents a sentence and contains the words and their corresponding tags.
- Converting each sentence in the test set to a PyTorch tensor of word indices, similar to what was done for the development set in the previous code example.
- Passing each sentence tensor through the trained model to obtain the predicted tag scores.
- For each sentence, selecting the tag with the highest score as the predicted tag, and storing these predicted tags in a list.
- Flattening the list of lists of predicted tags into a single list.
- Writing the original test data, along with the predicted tags, to an output file named **'test2.out'**. The predicted tag for each word in a sentence is written on a separate line, along with the corresponding sentence index and word itself.
- Overall, the code we write allows us to evaluate the performance of the trained model on an unseen test set and generate predictions for future use.