Assignment 4

In this assignment, we will reimplement the POS tagger of Assignment 2 with neural networks. We will use an LSTM sequence tagger and the Pytorch library.

There are a lot of tutorials on implementing neural tagging models on the Internet. You are welcome to read and watch them, but please submit code that is your own and not copied; plagiarism is not okay and will have consequences. Make sure you understand all the code you submit. I trust your honesty in this.

Be aware that training a neural network is time-intensive. On my computer, one epoch of training takes a few minutes, and a complete training run takes up to an hour. Consider developing and debugging your code on a subset of the training corpus until you believe it works correctly. Also, make sure you start early. If your own computer is not fast enough to train a neural network, feel free to use the department's compute servers or Google Colab.

Loading the training and test corpus

In a first step, you will load the training, development, and test data into a Pytorch `DataLoader`. We will use the German Universal Dependencies treebank as our dataset. It is annotated with POS tags, and indeed it is the same dataset that you used in Assignment 2, just in a different file format. Download the treebank from Github; you need the files `de_gsd-ud-{train|dev|test}.conllu`.

You will then have to load the corpus and convert it into a format that Pytorch can process. Data loading is a bit annoying and not very well documented, so I have implemented it for you. Download the file data.py and `import data` in your program. The function `data.load(training_filename, dev_filename, test_filename)` will return a tuple of the following values, in this order:

train_dataloader	DataLoader for iterating over the training data
dev_dataloader	DataLoader for iterating over the development data
test_dataloader	DataLoader for iterating over the test data
vocabulary	Vocabulary of words in the sentences in the data
tagset	Vocabulary of POS tags in the data
pretrained_embeddings	Pretrained fasttext word embeddings

Note that the first time you call `data.load`, it will download several gigabytes of pretrained word embeddings, which can take a while. The file then remains on your hard drive, so subsequent calls to `data.load` will be faster.

Each dataloader is an iterable over minibatches of (training, dev, or test) data. In our case, each minibatch contains a single instance (i.e., the batch size is 1); if `batch` is an element of the iterable, then `batch[0]` is of the form `(sentence, gold_tags)`, where `sentence`

and `gold_tags` are both LongTensors of shape `(sequence length,)`. The numbers in `sentence` are indices in the vocabulary; the numbers in `gold_tags` are indices in the tagset.

Have a look at the first instance in the train_dataloader and print out the original sentence and POS tags (in human-readable form, not the numbers).

Defining a neural model

Implement an LSTM-based POS tagger in Pytorch. In a first step, your tagger should assign pretrained word embeddings to the tokens in the sentence, using an Embedding layer. If your Embedding layer has the right size (the fasttext word embeddings have dimension 300), you can inject the pretrained embeddings into it as follows:

```
embedding.weight.data = pretrained_embeddings
embedding.weight.requires_grad = False
```

Your neural model should then pass these word embeddings as input to an LSTM layer and project the LSTM's hidden states into the dimension of the output vocabulary with a Linear layer of appropriate size. The result of applying your model to a sentence tensor of shape `(sequence length,)` should be a tensor of POS scores of shape `(sequence length, tagset size)`. Note that because there is only one sequence in each batch, you do not have to worry about padding the sequences in a batch to the same length, as some tutorials might suggest.

Use a cross-entropy loss to compare the predictions of your model against the `gold_tags`, and use the Adam optimizer to train your model. Print the mean training loss per epoch after each epoch to check that your model is actually learning something. The mean loss should be below 0.2. Note that the cross-entropy loss wants the raw scores ("logits") as input, before computing the softmax or log-softmax; this is why we did not implement a softmax layer after the Linear layer above.

Plotting learning curves

When developing neural networks, it is crucial to visualize the dynamics of the learning process. This will give you valuable information about whether your hyperparameters are right, whether there are bugs in your code, whether you are overfitting or underfitting, and whether you have trained your model for enough epochs.

Create a free account on comet.ml and follow the instructions to log data about your training run. You should at least log the training loss after each epoch. Submit a screenshot of a training run.

Evaluation

Write a function that evaluates the tagging accuracy of your model on a given sentence. Evaluate the accuracy of your model on the training and development corpus after each epoch, and have them plotted on comet.ml.

The training accuracy is of course an overly optimistic picture of the actual quality of your system's predictions. However, it is useful to judge whether your model is sufficiently powerful. Your aim should be to get a training accuracy that's close to 100%; if it is not, then you either have a bug in your code, or your model is too simple.

Once you have satisfied yourself that your system works well on the development data, evaluate it on the test data and report the tagging accuracy. Please resist the temptation to evaluate your tagger on the test data frequently; this will make your results less meaningful. It is okay to evaluate on the *development* data as often as you like, that's what it is there for.

You can either evaluate your trained model immediately after a training run, within the same Python file, or you can save your model to a file after training and then load it and do the evaluation in a separate program.

Hyperparameters

Your model has a lot of hyperparameters that you can experiment with. Play with different values and determine which one gives you the best results on the development set. There is no need to report results on this section of the assignment, but of course the thoroughness of your hyperparameter optimization will affect the final accuracy of your model.

Here are some hyperparameters you can manipulate:

- learning rate of the Adam optimizer
- dimensionality of the hidden states of the LSTM
- number of layers of the LSTM
- ordinary or bidirectional LSTM?
- if your LSTM has at least two layers, what dropout do you use?

Extensions

For extra credit, extend your model to improve its accuracy or the training speed. Consider, for instance, increasing the batch size and training on a GPU. Note that this is considerably more complicated than the batchsize 1 training above because the sequences in the same batch may not have the same length, and you will have to pad your sequences. Alternatively, try out other pretrained word embeddings for German. Or surprise me with a creative new idea. :)

Turn in before class on 2022-01-04 on Classroom.