Natural Language Processing – Exercise 3

Due: 21\12\2022 23:55

In this exercise, we will implement and compare 3 different models for a simple sentiment analysis task. We will use PyTorch to implement, train and test all 3 models.

General notes –

1. To make things clearer and easier, we provided a structured API for this exercise and some helper classes. These are meant for helping you, so whenever you feel limited by it, feel free to deviate from the API and change any of the given functions and classes (but make sure you explain your changes in the README).
2. You can use any of PyTorch modules for your implementation.
3. The code was tested on PytTorch 1.3. You can use other versions of PyTorch but it's your responsibility to modify the provided code if needed to make it work (and write it in the README). To load the pre-trained word embeddings, you will need to use the gensim package (you may find installation instructions here).

For your submission, please provide your code (including data_loader file if you changed it), README file and pdf file with all the results and answers.

1 The Dataset

The dataset we will use for the sentiment analysis task is the Sentiment Treebank dataset by Stanford.

This dataset consists of sentences, taken from movie reviews, and their sentiment value (a float between 0-most negative and 1-most positive). In this exercise, we will handle a binary classification task (i.e. predicting 0 for negative sentiment and 1 for a positive sentiment). For training, we will consider a sentiment value to be positive if it's ≥ 0.6, and negative if it's ≤ 0.4. That means we will discard examples with a sentiment that is between 0.4 and 0.6 (we implemented this for you).

The sentences in the dataset are presented as trees, where each sub-tree defines a sub-phrase and has its own sentiment value (as well as each word). For our purposes, we will train all models using the full sentences, as well as the sub-sentences.

We provided you with the file **data_loader.py** for the entire interaction you need with the dataset. This includes loading the data (both sentences and sub-sentences, as well as their binary label), and splitting the data into train, validation and test sets. Make sure you go over the data_loader.py file to understand its functionality and use to you.

Note about the data splitting - as described in the classification lecture, it is common (and crucial) practice when applying machine learning tools on datasets to first split them to train, validation (or dev) and test sets. The train set is the largest and used to optimize the model learned parameters. As we usually also have global parameters, either the model's or of the

optimization scheme's, which are not learned during training (those are called hyper-parameters), we use the validation set to choose their values based on some kind of search. The test set is only used to report unbiased results for the models with the chosen hyper-parameters. In this case we used an 80%,10%,10% split of the original dataset.

To get a better understanding of the models' advantages and disadvantages, we also provided code for extracting special subsets of examples out of the test set on which you'll also examine the model's performance. Those subsets include:

1. Negated polarity – All sentences in the test set where the sentiment polarity (positive / negative) of one of the main sub-phrases in the sentence is opposite of the whole sentence sentiment polarity. These are examples where a negation structure (of some kind) is in effect (as in the examples you saw in the sentiment analysis lecture). You can use $get\_negated\_polarity\_examples$ function to obtain the indices of those examples from the test set.
2. Rare words – We expect that some of our models will perform better than others on sentences with unseen or rare words. Therefore, we rank all test sentences by their maximal frequency of a word which has non-neutral sentiment value and take the 50 sentences ranked lowest. You can use $get\_rare\_words\_examples$ function to obtain the indices of those examples from the test set.

2 DataManager & OnlineDataset

We provided two additional utility classes for handling some of the functionalities needed to prepare the data for training in the PyTorch framework.

DataManager - the class handles the following tasks:

1. Loads the dataset sentences (with their labels)
2. Creates OnlineDataset instances which follow the PyTorch Dataset API, using some functionalities you will build in this exercise to convert the lists of tokens to valid inputs for each model.
3. Creates PyTorch DataLoaders which are used to iterate over batches of examples from the OnlineDatsets as PyTorch Tensors.

Please take your time to go through the class implementation and API, and PyTorch Dataloader and Dataset API's.

3 Model Descriptions

You will implement 3 different models:

Simple log-linear model:

This model will use a simple one-hot embedding for the words in order to perform the task.

The input to the model is the **average** over all the **one-hot** embeddings of the words in the sentence.

After receiving the average embedding, this model operates a single linear layer (sometimes called fully-connected layer, which is just a learned affine transformation), followed by a sigmoid in order to predict $p(y = positive|x)$.

Word2Vec log-linear model:

This model is almost identical to the simple log-linear model, except it uses pre-trained Word2Vec embeddings instead of a simple one-hot embedding.

The input to the model is the **average** over all the **Word2Vec** embeddings of the words in the sentence.

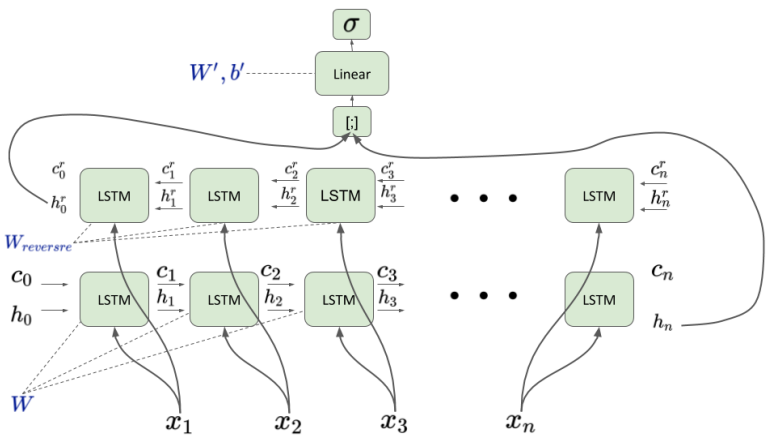
After receiving the average embedding, this model operates a single linear layer, followed by a sigmoid in order to predict the correct sentiment.

LSTM model:

The last model we're going to build is a more sophisticated LSTM model. Recall that the LSTM cell is a fancier RNN cell, that allows us to learn long-term dependencies in our data with a lesser risk of vanishing gradients. For the model we will build here, we use a bi-directional LSTM architecture, which is like the bi-RNN architecture seen in class but uses an LSTM cell.

In this model, each LSTM cell will receive as input the **Word2Vec** embedding of a word in the input sentence. We will then take the two hidden states of the LSTM layer (the last hidden state of both directions of the bi-LSTM layer) – and concatenate them. Later, we will put this concatenation through a linear layer and finally output the sigmoid of the result (representing again $p(y = positive|x)$) The dimension of the cell and hidden states of the LSTM will be 100.

You might find this figure helpful in understanding the suggested model:



4 Defining the loss for this task

As we've seen in class, when training a neural network (or any model) we train it to minimize some loss function. In this section we will present the loss function that will be used in this exercise.

A common loss function used in deep learning is the cross-entropy loss. This loss can be used for tasks of multi-class classification assuming that our model outputs distribution over the possible classes. We will use binary cross entropy, a version of this loss tailored for the binary case. Since we have only two classes, it's enough to predict the probability of one of

them and assume that the other probability is its complementary. A sigmoid can be used to map real values to the range [0,1]; it is thus common in the binary case to use sigmoid as the last activation layer in the model and compute the binary cross-entropy on it.

For numerical reasons, it works better to combine the sigmoid operation with the computation of the loss, thus PyTorch provides the **Binary cross entropy with logits** criterion which accepts as input the logits (fancy name for the inverse of the sigmoid) which are just the values before applying the sigmoid activation.

Recall that to try and minimize the loss function, we may use different algorithms. In our case, we choose to use the **ADAM** optimization algorithm.

5 Defining the accuracy measure for this task

Our accuracy measure for this task is rather simple – we simply measure the percentage of correctly classified sentences.

As we discussed in section 3, all of our models predict a number between 0 and 1 (due to the sigmoid function). However, the labels for our sentiment analysis task are 0 (for a sentence conveying negative sentiment) and 1 (for a sentence conveying positive sentiment). Thus, we need to round the output of the model to either 0 or 1.

Notice that here we do not discard values between 0.4 and 0.6.

1. **Implement** the method $binary\_accuracy(preds, y)$. This method will calculate the accuracy of the predictions given in $preds$, with respect to the true labels given in y.

6 Building a simple Log-Linear model.

In this step, we will build the aforementioned simple log-linear model, and test it.

1. Complete the implementation of the **LogLinear** class given to you. This class will serve us to define both the simple log-linear model, and the word2vec log-linear model. **Note:** although we use a sigmoid function to **predict** the sentiment value, it's not necessary to operate it in the **forward** method. We will use a loss function that performs the sigmoid function by itself, so be sure to implement the **forward** method without it.

As we mentioned in section 3, the simple log-linear model takes the average one-hot embedding of the words in the sentence as input. As you recall, a one-hot embedding of a word $w \in V$ is a vector $e \in \mathbb{R}^{|V|}$, such that $e_i = 1$ and $\forall j \in [|V|], j \neq i, e_i = 0$ (where i is the index of w in the vocabulary).

Therefore, to create a one-hot embedding, we must first assign an index to each word in the vocabulary.

2. Complete the $get\_word\_to\_ind(word\_count)$ method. This method will be used in order to create a mapping between words in the vocabulary, and their assigned indices.

After we have the index for each word in the vocabulary, we can move forward to create the one-hot embedding for each word, and to average all the embeddings of the words in the sentence.

3. Complete the following methods in order to assist you in calculating the average one-hot embedding:
   a. $\boldsymbol{get\_one\_hot(size, ind)}$
   b. $\boldsymbol{average\_one\_hots(sent, word\_to\_ind)}$

Now we're done with processing the data (in that case, turning it into an average of one-hot embeddings) and we can move on to defining and training our model.

We'll start by implementing the methods needed for the training process:

4. Implement the following methods:
   a. $train\_epoch(model, data\_iterator, optimizer, criterion)$. This method is in charge of running a single training epoch (epoch is a single pass over the entire training data).
   b. $evaluate(model, data\_iterator, criterion)$. This method evaluates the model on the test data it receives. **Note:** This method should be very similar to $train\_epoch$, except we don't want our model parameters to change while we're evaluating it.
   c. $train\_model(model, data\_manager, n\_epochs, lr, weight\_decay)$. This method initiates the entire training process. Be sure to save the train loss and accuracy, as well as the validation loss and accuracy, for each epoch in the training process.
   d. $train\_\log\_linear\_with\_one\_hot()$. This method should create all the objects needed for the training process, and run the training process.

Recall that we train our data using the train set, and validate it using the validation set. When we're done with parameter tweaking and testing out different settings, we need to test our final model – using the test set.

Since our model only outputs the logits, as we've discussed in section 3, we must use the $predict$ method defined in the $LogLinear$ class, in order to receive the predictions over the test set.

5. Implement the method $get\_predictions\_for\_data(model, data\_iter)$. This method will assist you in testing the model over the test set and the special subsets.

After implementing all the necessary methods, we are ready to train and test our model.

We will set the learning rate parameter value to be 0.01. We will train the model for 20 epochs with batches of size 64.

6. Train this model with weight decay: $w = 0.001$. Please provide:
   a. A plot of the train loss value, as a function of the epoch number (and another similar one for the validation loss). Plot both curves on the same graph (one curve for each parameter).

b. A plot of the train accuracy value, as a function of the epoch number (and another, similar one, for the validation accuracy). Plot both curves on the same graph.

P save and provide:

a. The test accuracy and loss.
b. The accuracy over each of the special subsets we've mentioned in section 1.

7 Building the Word2Vec log-linear model

In this section, you will implement the second model – the Word2Vec log-linear model.

As described in section 3, this model is pretty similar to the first model you've implemented. The difference is that the input for this model is the average **Word2Vec** embeddings of the words in the input sentence (rather than the one-hot embedding).

For that purpose, we will use pre-trained Word2Vec embeddings. Each embedding is a vector $e \in \mathbb{R}^{300}$. Note that not all words in our corpus will have a matching embedding. We will treat the words that have no embedding as unknown words.

To load these embeddings, we've provided you with the method $create\_or\_load\_slim\_w2v()$. Loading the embeddings might take a considerable amount of time, that can be spared by saving the dictionary mapping the known words in the corpus to their embeddings locally. Be sure to check out the documentation of the given method if you wish to save the embeddings locally and speed up the loading time.

Now, it's time to build the model. Luckily, the **LogLinear** class that you've implemented on section 5 will also serve us to define the Word2sVec log-linear model, so there's no need to create a different class for it.

The next step is, again, to process the data sentences to the required input for the model. That is, we wish to create the average **Word2Vec embedding for the words in the sentence.** As we mentioned earlier, we will not find embeddings for some of the words in the corpus. We would still need to map these words to some vector in $\mathbb{R}^{300}$. In this case, we will map the unknown words to the zero vector.

1. Implement the method$get_w2v_average(sent, word_to_vec, embedding_dim)$.
   This method returns the average **Word2Vec** embedding of the given sentence. (Average without the unknowns).

We may also see that the training process is identical for both this and the previous model, so there's no need to redefine the basic train methods you've implemented last time, except for the wrapping method.

1. Implement the method $train\_\log\_linear\_with\_w2v()$. This method should create all the objects needed for the training process, and run the training process.
2. **For this section, please follow the same parameter-dependent model training we've seen in section 6. Also, save and provide the same plots and values.**

8 Building the LSTM model

In this (last) section, we will implement the LSTM model described in section 3.

As in section 4, we should start by defining the class for this model.

1. Implement the class $LSTM$. Please follow the architecture of the model as described above. We will also use dropout regularization over the input to the linear layer for training. You can use PyTorch's provided LSTM module for this class implementation.

Recall this model's description from section 4. The input for each bi-LSTM cell is the embedding of a single word. For that purpose, we again have to load the pre-trained Word2Vec embeddings, and use it to map each word in the sentence to its embedding.

The input to our model should be a sentence, after each word in it was mapped to its embedding. One thing to note here, is that since we perform our training in batches, we must make sure that all sentences in the batch are of the same size (i.e. have the same number of words). To simplify things, we will require that all sentences in the set to have the same size. Clearly, not all of the sentences in the dataset are of the same size. Hence, our solution is to cut\pad all sentences up to a pre-determined length. Specifically, we set the pre-determined length to be 52. So:

- For all sentences with **More** than 52 words: we will map each of their first 52 words to their embeddings and ignore all other words.
- For all sentences with **Less** than 52 words: we will map each of their x words to their embeddings. These embeddings should be followed by $52 - x$ more 0-vectors, to make sure the sentence is of length 52.
- For all sentences with exactly 52 words: We will map each of its 52 words to its embedding.

2. Implement the method $sentence\_to\_embedding(sent, word\_to\_vec, seq\_len, emedding\_dim)$. This methods purpose is to map the given sentence to its word embeddings.

Lucky for us, the training methods are, again, similar to the ones you've already defined for sections 5,6 and so there's no need to write new ones. So, all that's left to be done is to again define the wrapping training method:

3. Implement the method $train\_\text{lstm}\_with\_w2v()$. This method should create all the objects needed for the training process, and run the training process.

Now, we just have to train the model:

4. **Train the model once with learning rate 0.001, weight decay value of 0.0001, dropout probability of 0.5 and the same batch size as before for 4 epochs.**
5. **Please provide: the same plots and results as in sections 6, 7.**

9 Comparing the different models

In this section, we will analyze the results you've received for each of the models.

1. Compare the results (test accuracy, validation accuracy) you've received for the simple log-linear model, and the Word2Vec log-linear model. Which one performs better? Provide a possible explanation for the results you have.
2. Compare the latter results with the results of the LSTM model. Which one performs better? Provide an explanation for the results you received.
3. Last, compare the results that all the models had on the 2 special subsets of sentences we've provided you. For each subset, state the model that has the highest result (and the lowest result) and provide a possible explanation for these results.

10 Tips & recommendations

1. Be sure to check PyTorch's documentation for each of the layers\functions you're using. It's specifically important to figure out what each method expects to receive, and outputs.
2. Watch out for dimension and type issues and misfits – you might need to reshape or cast some of your input in some methods. Be careful when reshaping – make sure you do not shuffle the values.
3. Use *with torch.no_grad():* during evaluation. In general, when you don't need gradiants, you should aviod calculating them to save time.