NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

Speech and Natural Language Processing
Spring Semester 2022-23

3rd Lab Preparation

Introduction

The purpose of this lab is to implement a model for processing and classification of texts, using deep neural networks (DNN) and pre-trained large language models (LLMs). For the development of the DNN models you should use the Pytorch library [1]. At first, using pre-trained vector representations of words, you are asked to create representations for each text. Then, you will use the representations of the texts to make the classification. The goal is to train models that can perform sentiment analysis in sentences. You are provided with 2 datasets (which already exist in the exercise repository).

- Sentence Polarity Dataset[2] [Pang and Lee, 2005]. This Dataset contains 5331 positive and 5331 negative film reviews, from Rotten Tomatoes and is a binary-classification problem (positive, negative).
- Semeval 2017 Task4-A[3] [Rosenthal et al., 2017]. This Dataset contains tweets that are categorized in 3 classes (positive, negative, neutral) with 49570 training examples and 12284 testing examples.

Development Environment

At first you should set up the development environment on your computer. First clone the repository: `https://github.com/slp-ntua/slp-labs/tree/master/lab3`. Then download pretrained word vectors of your choice in the folder */embeddings* of the project. Indicatively we suggest that you download the GloVe[4] embeddings, as embeddings are available in low dimensions, which means fewer computational requirements. Alternatively you could use the FastText[5] embeddings, which, however, are only available in 300 dimensions. Depending on the dataset you will notice that different embeddings achieve better results. For example, in the dataset Semeval 2017 Task4-A, you will see better results with Twitter Glove Embeddings, as they are trained on tweets. The choice is yours and you will not be evaluated for the performance of your model, but on the correctness of your answers.

You should install certain libraries to implement the exercise. It is recommended to install them in an isolated environment so that there is no undesirable conflict with other libraries on your computer. The easiest solution is to use the tool `conda`. If you have not already installed it on your computer, download the correct version of Miniconda for **version 3** of Python and install it by following these instructions. Finally, create a new environment[6] and enable it as follows:

```
conda create -n slp3 python=3.8
conda activate slp3
```

[1] `https://pytorch.org/`
[2] `http://www.cs.cornell.edu/people/pabo/movie-review-data/`
[3] `http://alt.qcri.org/semeval2017/task4/index.php?id=data-and-tools`
[4] `https://nlp.stanford.edu/projects/glove/`
[5] `https://fasttext.cc/docs/en/english-vectors.html`
[6] `https://conda.io/docs/user-guide/tasks/manage-environments.html`

Install Pytorch by following these instructions and then all the remaining libraries with the command:

```
pip install -r requirements.txt
```

Familiarise yourself with the structure of the project and carefully study the code already implemented. Red color indicates the files that you should edit as part of the exercise:

```
/datasets           folder with training data
/embeddings         folder with pretrained word embeddings
/utils              folder with utility python scripts
config.py           settings regarding the project
dataloading.py      preprocessing and preparation of examples
main.py             main script - starting point
models.py           contains models with neural networks
training.py         contains functions for model training
```

The purpose of this task is to fill in gaps in the code. Many of the trivial parts of the exercise have already been implemented and you are called upon to intervene at specific points in the code, depending on the question. In the majority of questions, you should fill in only a few lines. By opening the file **main.py**, you will find that the steps of loading the training data, as well as the pretrained word embeddings have already been implemented for you. The position in which you should implement the solution of each question in the source code will be marked with **FILE:EXi**, where **FILE** is the file and **EXi** the question ID, which will be in comments in the code[7].

1 Data Preprocessing

In this step you need to process the data so that you can train the neural network. For better handling of the training data you will use the tools provided by Pytorch (Dataset and Dataloader classes etc.[8]), inheriting the corresponding classes and making your own, depending on the needs of the exercise. The class `torch.utils.data.Dataset` converts each example to the form required for the training of the neural network and the class `torch.utils.data.Dataloader`, uses an object of the class `Dataset` to convert its examples into torch Tensors and organize them into mini-batches. The extension of the `Dataset` class that you will make, will contain the variables with the examples and labels of each data set, as well as the methods for processing and preparing them.

1.1 Label Encoding

Initially, training examples are in the form of text (position, neutral, negative...).You should encode them so that each class corresponds to a specific number. The mapping should be the same for the training and the test set.

Hint: Use `LabelEncoder`[9] of scikit-learn.

Question 1: Fill in the blanks in the position `main.py:EX1` and print the first 10 labels from the training data and their mappings in numbers.

[7] Just search the code based on the ID and you will find all the occurrences

[8] https://pytorch.org/docs/stable/data.html

[9] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html

1.2 Tokenization

In this process, a text is converted from a sequence of characters to a sequence of tokens or terms, such as words, punctuation, numbers, etc. In the file `dataloading.py` the class `SentenceDataset` has been declared, which extends the class `torch.utils.data.Dataset`. Run the tokenization at the initialization of the Dataset for all data and keep the processed data in a variable in the class. You can split each text based on the space character, or perform some more sophisticated lexical analysis according to the particularities of the Dataset (e.g. NLTK, spaCy, ekphrasis).

Question 2: Fill in the blanks at position `dataloading.py:EX2` and print the first 10 examples of training data.

1.3 Encoding of Examples (Words)

This is the final step in which you should prepare each example in the appropriate form for training the neural network. It is required to implement the following:

1. each term (token - term) must be mapped to a number so that the Embedding Layer can match it to the correct embedding. The function that loads the pretrained word embeddings `load_word_vectors` returns a python dictionary which has the form `word:id`. Use it. If a term does not exist in dictionary then you should match it to the term `<unk>`.

2. all examples should be the same length. This is required so that the linear algebra can be performed, such as matrix multiplication. For this reason you should select a maximum length of sentences and fill in zero elements to the shorter-length sentences (zero-padding) or remove words that exceed the selected length. You can print the distribution with the lengths of sentences in the training set and choose a length that covers the majority by ignoring outliers.

The method `__getitem__` of the class `SentenceDataset` must return the following: 1) the encoded form of a sentence, 2) the id of the label 3) the **real** length of the sentence (that is, excluding zero elements).

Example of encoding a sentence, for max_length = 8:

```
dataitem = "this is a simple example", label = "neutral"

Return values:
example = [  533  3908  1387   649   88     0     0     0]
label = 1
length = 5
```

Question 3: Implement the method `__getitem__` of the class `SentenceDataset` (at position `dataloading.py:EX3`) and print 5 examples in their original form and in the form that the class `SentenceDataset` returns.

2 Model

In this step you should design a neural network, which 1) will create a continuous vector representation for each term in a sentence using an embedding layer, 2) will create a vector representation for the entire text of an example, 3) will categorize the text based on its representation in the correct class. Model source code is in the file `models.py` and you should implement the methods `__init__` and `__forward__` of the class `BaselineDNN`. In the method `__init__` you will declare the network layers and initialize their weights. In the method `__forward__` you will define the transformations of the input data to produce the model's final output (forward pass).

2.1 Embedding Layer

First, you will use an embedding layer to project each term / word of a text in a continuous space (embedding space). The embedding layer places the words in the space according to the relationships they have between them. Words that co-occur often will be close to each other. The vector corresponding to the position of each word is called word representation or word features or word embedding. The embedding layer weights can be initialized at random values and you can update them when training the model or you can initialize them from pretrained word embeddings.
Implement the following:

- Create an embedding layer[10].
- Initialize network weights from pretrained word embeddings. The function `load_word_vectors` that loads the pretrained word embeddings returns the two-dimensional matrix with weights (`shape: num_embeddings, emb_dim`). Use it.
- Freeze the embedding layer, that is, define that the network weights will not be further updated when training the model.

Briefly answer the following:

- Why do we initialize the embedding layer with pretrained word embeddings?
- Why do we keep the weights of the embedding layer frozen during training?

Question 4: Fill in the gaps at the positions `models.py:EX4` and answer the questions above.

2.2 Output Layer(s)

For classification, you should project the representations of texts at the space of classes. That is, if the final representations are 500 dimensions and the classification problem has 3 classes, then the last layer should make a projection $R^{500} \rightarrow R^3$.
Implement the following:

- Create a layer with a non-linear activation function (e.g. ReLU, tanh...), which will learn a transformation of the representation of every example.
  *Suppose the dimensions of the representations of the examples are the dimensions of the word embeddings. For the output of the layer, choose the dimensions you want.
- Create the last layer that projects the final representations of texts to the classes.

Briefly answer the following:

- Why do we put a non-linear activation function in the penultimate layer? What difference would it make if we had 2 or more linear transformations in a row?

Question 5: Fill in the gaps in the positions `models.py:EX5` and answer the questions above.

[10] https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html#torch.nn.Embedding

2.3 Forward pass

After you have designed the layers that your model will use, the last step is to design how the network will transform the input data to the corresponding outputs (predictions). The input to the model will be a mini-batch, with dimensions (`batch_size, max_length`). In this question you are asked to create a very simplistic model that will perform the following transformations:

1. Project the words of each sentence with the embedding layer you created, where each term (word) will correspond to a vector. The input will have dimensions (`batch_size, max_length`) and the output (`batch_size, max_length, emb_dim`).
2. Create a representation for each sentence. The sentences are variable in length, but all representations of the texts should be in the same space (the same dimensions). You can create a simple representation by calculating the average (center of gravity) of the word embeddings in one sentence. Thus all the sentences will have a vector representation with the same dimensions as the word embeddings.
3. Apply the non-linear transformation you designed on the previous question to each representation and create new deep-latent representations.
4. Project the final representations to the space of classes.

Briefly answer the following:

- If we consider that every dimension of the embedding space corresponds to an abstract concept, you can give an intuitive interpretation of what the representation you made (center of gravity) describes.
- Report possible weaknesses of this approach to represent texts.

Hint: To calculate the average word embeddings of a sentence, you must consider and **exclude** zero-padded timesteps. For this reason you should use the actual lengths of each sentence you have calculated in the method `__getitem__` of the class `SentenceDataset` and which you have passed as an input to the method `__forward__` of the model. For example, in the sequence $[3, 5, 2, 8, 0, 0, 0, 0]$ the average should be 4.5 and not 2.25, since the actual length of the sequence (as meant under the problem) is 4 and not 8.

Question 6: Fill in the gaps in the positions `models.py:EX6` of the method `__forward__` and answer the questions above.

3 Training procedure

In this step you need to implement the network training procedure, that is, you will organize the examples into mini-batches and run stochastic gradient descent to update the network weights.

3.1 Loading Examples (DataLoaders)

After implementing the class `SentenceDataset`, in order to convert the examples into the appropriate form, you should design how the neural network will be trained from the corresponding examples. Use the Dataloader class and make an instance for each dataset. Give a brief answer to the following:

- What are the consequences of small and large mini-batches in model training?
- We usually shuffle the order of mini-batches in the training data in every epoch. Can you explain why?

Question 7: Fill in the gaps in the positions `main.py:EX7` and answer the questions above.

3.2 Optimization

To optimize the model you must define the following:

1. Criterion. If the classification problem has 2 classes then use the `BCEWithLogitsLoss`, while if it has more then use the `CrossEntropyLoss`. Beware of the dimensions of the output layer of the neural network depending on the criterion you choose.
2. Parameters. Select the parameters that will be optimized. Caution, as we do not want to train all the parameters of the network.
3. Optimizer. Select an optimization algorithm. It is recommended to use an algorithm that automatically adjusts the speed of updating the weights (Adam, Adagrad, RMSProp...).

Question 8: Fill in the gaps in the positions `main.py:EX8`.

3.3 Training

The last step for training the model is to implement the methods for the training and evaluation of each mini-batch. The function `train_dataset` is called for each batch in one epoch, it provides the training data to the model, it calculates the error and updates the network weights with the backpropagation algorithm. Similarly, the function `eval_dataset` is called at the end of each epoch to evaluate the model.

Hint: To calculate network predictions (most probable class) during evaluation, where you need to calculate the argmax of the posteriors, you can use the function max[11]. If the problem is a binary-classification one then you are interested in whether the logit is greater than zero (probability above 0.5).

Question 9: Fill in the gaps in the positions `training.py:EX9`.

3.4 Evaluation

After you have completed the development of the model, the final task is to evaluate its performance. You can decide on the number of epochs that you will train the model. This number may be a fixed predifined value, or you can stop training when the loss on the test set stops decreasing.

Question 10: For each of the 2 datasets given to you, report the model's performance on these metrics: accuracy, F1_score (macro average), recall (macro average). Also create graphs that will show the model training curves (training και test loss) per epoch.

4 Classification using an LLM

Large Language Models like ChatGPT [12] are being made available and can be used on a great variety of linguistic tasks and, thus, on sentiment classification of texts.

Question 11: For each of the two datasets, select at least 20 samples from each category and use ChatGPT for their sentiment classification. To that end, provide proper prompts to the model (experiment with various alternatives). Measure the performance of the model. Did it make any mistake; By providing the proper instructions/prompts, try to derive insights with regards to the reasons it classified each sample to the respective category. Were some words more important than others in terms of the sentiment of the sentence?

[11] https://pytorch.org/docs/stable/generated/torch.max.html#torch.max
[12] https://chat.openai.com/

References

[Pang and Lee, 2005] Pang, B. and Lee, L. (2005). Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the ACL*.

[Rosenthal et al., 2017] Rosenthal, S., Farra, N., and Nakov, P. (2017). SemEval-2017 task 4: Sentiment analysis in Twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation*, SemEval '17, Vancouver, Canada. Association for Computational Linguistics.

Appendix – Further material

Official guides

You can start studying from the following general guides:

- Deep Learning with PyTorch: A 60 Minute Blitz
- Learning PyTorch with Examples

And continue with those who focus on NLP problems:

- Introduction to PyTorch
- Deep Learning with PyTorch
- Word Embeddings: Encoding Lexical Semantics

Guides from third parties

The guides from Stanford's "CS230 Deep Learning" course are excellent:

- Introduction to Pytorch Code Examples - An overview of training, models, loss functions and optimizers
- Named Entity Recognition Tagging - Defining a Recurrent Network and Loading Text Data

Furthermore:

- The article "PyTorch - Basic operations" is a very good point of reference with the basic operations you can do with tensors in Pytorch.
- The list of implemented models "Pytorch: Lists of tutorials & examples" may be useful to you.
- And the guide "Writing Custom Datasets, DataLoaders and Transforms" for questions regarding the preparation of data.

Finally, we recommend testing individual features using a separate script to easily and quickly control their behavior. As in the following example of an embedding layer:

```
import torch
import torch.nn as nn

batch_size = 16
max_length = 8
n_embeddings = 1000
embedding_size = 50
```

```
# create a batch of random sequences of token ids
x = (torch.rand(batch_size, max_length) * n_embeddings).long()
print(x.shape)

embed_layer = nn.Embedding(num_embeddings=n_embeddings, embedding_dim=embedding_size)

embeddings = embed_layer(x)
print(embeddings.shape)
```