NATIONAL TECHNICAL UNIVERSITY OF ATHENS
FACULTY OF ELECTRICAL & COMPUTER ENGINEERING

Speech and Natural Language Processing
Spring Semester **2024-25**

Preparation of the 2nd Workshop

Introduction

The aim is to implement a model for text processing and categorization using Deep Neural Networks (DNN) and pre-trained Large Language Models (LCLMs). To develop DNN models you should use the PyTorch[1] library. Initially, using pre-trained vector word embeddings, you are asked to create representations for each text. Then, you will use the text representations to do the categorization. The goal is to train models that can perform sentiment analysis on sentences. You are provided with 2 datasets (already contained in the exercise repository).

- Sentence Polarity Dataset[2] [Pang and Lee, 2005]. This dataset contains 5331 positive and 5331 negative movie 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 into 3 classes (positive, negative, neutral) with 49570 training examples and 12284 evaluation examples.

Development Environment

First you need to set up the development environment on your computer. First download the repository: https://github.com/slp-ntua/slp-labs/tree/master/lab3. Then download pre-trained word vectors of your choice in the */embed- dings* folder of the project. As an indication, we recommend downloading the GloVe[4] embeddings, as there are embeddings available in low dimensions, which means less computational requirements. Alternatively you could also use FastText[5] embeddings, but these are only available in 300 dimensions. Depending on the dataset you will notice that different embeddings achieve better results. For example, in the Semeval 2017 Task4-A dataset, you will see better results with the Twitter Glove embeddings, as they are trained on tweets. The choice is yours and you will not be evaluated on the performance of your model, but on the correctness of your answers.

You will need to install some libraries to implement the exercise. It is recommended that you install them in a separate environment, so that there is no undesirable effect on the other libraries on your computer. The easiest solution is to use the conda tool. If you don't already have it installed on your computer, download the correct version of Miniconda for Python version **3** and install

[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

following the instructions. Finally, create a new environment[6] and activate it as follows:

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

Install PyTorch following the instructions and then all the other libraries with the command:

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

Familiarize yourself with the project structure and carefully study the code that has already been implemented. The red color indicates the files you will have to edit in the exercise:

`/datasets`	folder with training data
`/embeddings`	folder with the pre-trained `word embeddings`
`/utils`	folder with auxiliary `Python scripts`
`config.py`	settings on the `project`
`dataloading.py`	preprocessing and preparation of examples
`main.py`	main `script` - starting point
`models.py`	contains the models with the neural network(s)
`training.py`	contains the functions for training the models

This particular task has the logic of filling in gaps in the code. Many of the trivial parts of the exercise have already been implemented and you are asked to intervene at specific points in the code, depending on the question. In the majority of the questions, only have to fill in a few lines. By opening the **main.py** file, you will see that the step of loading the training data, as well as the pretrained word embeddings, has already been implemented for you. The location where you should implement the solution to each query in the source code will be marked **FILE:EXi**, where **FILE** is the file and **EXi** is the query ID, which will be in comments in the code[7].

1 Data pre-processing

In this step you need to process the data so that you can then train the neural network. To better manage 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 `torch.utils.data.Dataset` class converts each example into the format required for neural network training, and the `torch.utils.data.Dataloader` class, uses an object of the `Dataset` class to convert its examples into torch Tensors and organize them into mini-batches. The extension to the `Dataset` class you build will contain the variables with the examples and labels of each dataset, as well as the methods for processing and initializing them.

1.1 Coding of **Labels (Labels)**

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

Help: use scikit-learn's `LabelEncoder`[9].

[6] https://conda.io/docs/user-guide/tasks/manage-environments.html

[7] just search the code of the file by ID and you will find all the references

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

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

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

1.2 Verbal Analysis (Tokenization)

In this process, a text is converted from a sequence of characters into a sequence of tokens or , such as words, punctuation marks, numbers, etc. The `SentenceDataset` class is declared in the `dataloading.py` file, which extends the `torch.utils.data.Dataset` class. Perform tokenization when initializing the Dataset on all its data and keep the processed data in a variable in the class. You can split each text based on the character of the space, or perform some more sophisticated verbal analysis depending on the specifics of the dataset (such as NLTK, spaCy, ekphra- sis).

Task 2: Fill in the blanks in `dataloading.py:EX2` and print the first 10 examples from the training data.

1.3 Encoding Examples (Words)

This is the final step, in which you should prepare each example in the appropriate format for neural network training. You are required to implement the following:

1. each token term must be mapped to a number so that the embedding layer can map it to the correct vector representation (word embedding). The function to load the pre-trained word embeddings `load word vectors` returns a python dictionary which has the format `word:id`. Use it. If a term does not exist in the dictionary, then you must assign it to the <unk> term.

2. all examples should have the same length. This is required to be able to perform linear algebra operations, such as matrix multiplication. For this reason, you should select a maximum sentence length and zero-padding shorter sentences or removing words that exceed the selected length. You can print the distribution of sentence lengths in the training set and choose a length that covers the majority, ignoring outliers.

The `__getitem__` method of the `SentenceDataset` class must return the following: 1) the encoded form of a sentence, 2) the id of the tag 3) the actual length of the sentence (i.e. excluding null elements).

Example of coding a sentence, for a maximum length of= 8:

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

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

Task 3: Implement the `__getitem__` method of the `SentenceDataset` class (location `dataloading.py:EX3`) and print 5 examples in their original form and as returned by the `SentenceDataset` class.

2 Model

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

2.1 embedding layer

First, you will use an embedding layer, to display 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 between them. Words that appear together frequently will be close to each other. The vector corresponding to the position of each word is called the word representation or word features or word embedding. The embedding layer weights can be initialized with random values and updated during model training or initialized from pre-trained word embeddings.
Implement the following:

- Create an embedding layer[10].
- Initialize the network weights from the pre-trained word embeddings. The load function of the pre-trained word embeddings `load word vectors` returns the two-dimensional matrix with the weights (`shape: num embeddings, emb dim`). Use it.
- Freeze the embedding layer, i.e. declare that the network weights will not be further updated during model training.

Please answer the following briefly:

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

Task **4**: Fill in the blanks in `models.py:EX4` and answer the questions above.

2.2 Output Layer(s)

For categorization, you need to view the text representations in the class space. That is, if the final representations are 500 dimensional and the categorization problem has 3 classes, then the last layer should project $R^{500} \rightarrow R^{3}$.
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 each example.
  *Assume that the dimensions of the example representations are the same as the dimensions of the word embeddings. For the output of the layer, you choose whatever dimensions you want.
- Create the last layer which displays the final representations of the texts in the classes.

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

Please answer the following briefly:

- 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?

Task **5**: Fill in the blanks in `models.py:EX5` and answer the questions above.

2.3 Forward pass

Once you have designed the layers that your model will use, the last step is to design how the network will transform the input data into 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 simple model that performs the following transformations:

1. View 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 of variable length, but all text representations should be in the same space (same dimensions). You can create a simple representation by calculating the average (center of gravity) of the word embeddings in a sentence. Then all sentences will have a vector representation with dimensions equal to the dimensions of the word embeddings.
3. Apply the non-linear transformation you designed in the previous question to each representation and create new deep-latent representations.
4. View the final representations in class space. Briefly answer

the following:

- If we assume that each dimension of the embedding space corresponds to an abstract concept, can you give an intuitive interpretation of what the representation you have built (center-of-mass) describes?
- Indicate possible weaknesses of this approach to representing texts.

Help: to calculate the average of the word embeddings of a sentence, need to take into account and exclude zero-padded timesteps. For this reason you should use the actual lengths of each sentence, which you have calculated in the method `__getitem__` of the `SentenceDataset` class and which you have passed as input to the `__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 understood in the context of the problem) is 4 and not 8.

Task **6**: Fill in the blanks in the `models.py:EX6` locations in the `__forward__` method and answer the questions above.

3 Training process

In this step you will need to implement the network training process, i.e. you will organize the examples into mini-batches and perform stochastic gradient descent to update the network weights.

3.1 Loading Examples **(DataLoaders)**

After you have implemented the `SentenceDataset` class to convert the examples into the appropriate format, you must then design how the neural network will be trained from the corresponding examples. Use the DataLoader class and build a snapshot for each dataset. Provide a short answer to the following:

- What are the implications of small and large mini-batches for model training?
- We usually shuffle the order of the mini-batches in the training data at each epoch. Can explain why?

Task **7**: Fill in the blanks in `main.py:EX7` and answer the questions above.

3.2 Optimization

To optimize the model you need to specify the following:

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

Task **8**: Fill in the blanks in `main.py:EX8`.

3.3 Education

The last step in training the model is to implement the methods for training and evaluating each mini-batch. The `train dataset` function is called for each batch in an epoch, gives the training data to the model, calculates the error, and updates the network weights with the backpropagation algorithm. Similarly, the `eval dataset` function is called at the end of each epoch to evaluate the model.

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

Task **9**: Fill in the blanks in `training.py:EX9`.

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

3.4 Evaluation

Once you have completed the development of the model, the final task is to evaluate its performance. The number of seasons you train the model can be decided by you. The number can be a default value, or you can stop training when the error (loss) in the test set stops decreasing.

Task **10**: For each of the 2 datasets provided, report the performance of the model on the metrics: accuracy, F1 score (macro average), recall (macro average). Also, create graphs showing the training and test loss curves of the model by season.

4 Categorisation using **LLM**

Large linguistic models such as ChatGPT [12] are now available and can be used for a wide range of language-related tasks, and thus for categorising the sentiment of a text.

Task **11**: For each of the 2 datasets, select at least 20 texts from each category and use ChatGPT to identify the sentiment. Provide appropriate prompts to the model for the above purpose (experiment with different alternatives). Measure the performance of the model. Did it make a mistake? Provide appropriate prompts and try to extract information about the reasons why each text was classified in its respective category. Were some words more important than others in terms of the sentiment of the sentence?

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.

Annex - Supporting Material

Official Guides

You can start your study 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

[12] https://chat.openai.com/

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

- 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

In addition:

- The article "PyTorch - Basic operations" is a very good 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.
- As well as the guide "Writing Custom Datasets, DataLoaders and Transforms" for questions about data preparation.

Finally, we recommend testing individual functions using a separate script, so that you can easily and quickly check their behaviour. As in the following example concerning 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)
```