Deep Learning Workshop, 27.03 – 30.03.2023

Take Home Assignment: Recurrent Neural Networks (RNN)

Closing and call for feedback

Thank You for Attending the Workshop. We are pleased by your interest and engagement in the material presented, and we hope that you found the information and discussions to be valuable and informative.

As we want to improve the quality and effectiveness of the provided material in the future, we would greatly appreciate any feedback you may have. Your input is essential in helping us identify areas where we can improve and make changes. Please take 2-3 minutes to fill out a feedback form, which you can find under **Feedback Form** (if you have not done it already). Alternatively, you can also reach out to us via email, and share your thoughts, comments, and suggestions.

Additionally, if you have found a better solution or have implemented one of the bonus questions, we would love to see your work and potentially integrate it into our materials. You can do this by creating a pull request on our GitHub repository or contact us via email.

Once again, thank you for your time and engagement. We value your participation and feedback.

Take Home Assignment

The main goals of this last programming/take home assignment are to:

a) Perform a data preprocessing pipeline (Tokenization/Segmentation, One-hot-encoding) for a Natural Language Processing (NLP) task by hand.

b) Compare the performance of a Recurrent Neural Network (RNN) and a Long short-term memory (LSTM) to analyze movie reviews (Sentiment Analysis).

Disclaimer: The way we perform the data preprocessing in this task is for educational purposes only and thus neither efficient nor a state-of-the-art method.

We will provide solution proposals for all of the exercises as well as for the first two bonus questions one week the workshop.

Theoretical Introduction

In this programming assignment, we need the following theoretical components that will be described briefly. The explanations are based on the lecture notes of the course "Informationsmanagement" held by Dr. Thomas Arnold at TU Darmstadt in the summer term 2022.

Text Segmentation/Tokenization Tokenization is the task of segmenting an input stream into an ordered sequence of tokens. Usually tokens correspond to inflected word forms. A system for segmenting streams is called tokenizer. Text segmentation helps to identify the structure of the text and can improve the performance of downstream tasks such as named entity recognition, part-of-speech tagging, and sentiment analysis. There are several methods for text segmentation, including rule-based methods, statistical methods and machine learning-based methods. Rule-based methods (which we will use) utilize a set of predefined rules or heuristics to segment text, such as punctuation marks, capitalization, and whitespace.

Embedding Layer (PyTorch) An embedding layer (in PyTorch) is a module that converts discrete input data (such as words or integers) into a continuous vector representation, known as an embedding. These embeddings can then be used as input for a neural network, which can learn to extract useful features from the data. The embedding layer is typically used for tasks such as NLP and Computer Vision (CV). You can think of an embedding as a look-up table that can be learned during the training process, see also here for an example.

Programming

In this task we will implement two classifiers (RNN [Jor86]; [RHW86] and LSTM [HS97]) to predict whether movie reviews are positive or negative. We use the "Large Movie Review Dataset" [Maa+11] which contains 25k reviews for training and 25k reviews for testing. As using the raw data would require a lot of preparation and structural formatting, we will work with an already preprocessed version that you can download from Kaggle and is also available in the `./data` folder.[1] The original dataset is available here and provides additional unlabeled data.

In order to run the segmentation of our input texts, we need the *Natural Language Toolkit*[2] (NLTK) as an additional Python package. Please execute the following commands to install NLTK and download the required stopword-database:

```
pip install nltk
python
>>> import nltk
>>> nltk.download("stopwords")
```

1) **Data Preparation: Why did the text file break up with the tokenizer? Because it couldn't handle the segmentation.**
   In this exercise we will work with the file `rnn.py` and `nlp_utils.py`. Open the folder `./data`, you will find a `.csv`-file containing all reviews and the corresponding labels in a tabular format. Familiarize yourself with the data: Load the table inside a second script, analyze the distribution of the labels and the lengths of the reviews. As usual, but especially in NLP tasks, this is a crucial step.
   Now, to start with the first task, open the file `nlp_utils.py`.

   a) First, we want to perform the preprocessing of a single string. Implement the `preprocess_str` function. Define regular expressions using the `re` module[3] to remove all non-word characters, replace multiple whitespaces with no spaces and replace all digits with no spaces.

   b) Next, implement the `_most_common_words` function, generate a vocabulary with all occurring words and their corresponding frequencies, i.e. the `Counter` class of the `collections` module[4]

[1]Our dataset will consist of one big table and we will have to manually split the data by ourselves.
[2]See https://www.nltk.org/.
[3]See https://docs.python.org/3/library/re.html.
[4]See https://docs.python.org/3/library/collections.html?highlight=counter#collections.Counter

might be helpful to hash the words and keep track of their frequencies. Use the `preprocess_str` function for each word first and ensure that you only add non-stopwords. You obtain a set of stopwords by using the NLTK package:

```python
from nltk.corpus import stopwords
stop_words = set(stopwords.words("english"))
```

Only return the `top_n_words` in order to reduce the dimensionality of our input data. You can check later how this constraint influences the quality of your predictions.

c) Now we can perform the tokenization step by implementing the function `tokenization`. Call your `_most_common_words` function first and generate an index with the words, i.e. number each word of the vocabulary.

   - Iterate through the raw data once again: For each review remove words that are not in the vocabulary and generate a list of indices for words contained in the vocabulary. Each review is originally one large string and should be afterward represented by a list of numbers, each representing themselves a word contained in the vocabulary. For instance, the sentence "I really like RNNs." with an index for the most common words of the form `{"really": 0, "like": 1}` should be transformed to the list `[0, 1]`.
   - Create a `numpy` array for the labels. Iterate through the labels and convert `"positive"` to a value of 1.0 and `"negative"` to a value of 0.0. It is really important that the label's data type is `dytpe=np.float32`. Otherwise we will end up with a wrong conversion when switching to PyTorch tensors and the loss function will throw an error.

   Return the list (of lists containing indices) in its raw format, the labels as a `numpy` array and also add the underlying index (which maps words to indices and vice versa) as a return value.[5]

d) Finally, we can finish the data preparation task by completing the `add_padding` function. The function takes the tokenized reviews as an input and performs an (optional) padding operation, i.e. longer review will be cut off and shorter ones are filled up with zeros. The parameter `target_length` indicates which size each review should have after the operation.
   Start by creating a large `numpy` matrix filled with zeros of the shape `len(sentences)`$\times$`target_length`. For each review perform a (optional) clipping and add the vector as a column into the matrix.

e) Combine steps a) to d) in the `prepare_data` function which will be called by our `DatasetIMDB` class.

As already mentioned in the beginning, the method presented above is not optimized for efficiency and does not scale for large corpora and/or large datasets. We provide the `@log_time` decorator for you to observe bottlenecks in your code and determine potential locations for optimization. Consider working on some of the bonus tasks if you are interested in more advanced preprocessing methods.

Open the file `rnn.py` and take a look at the template of the `DatasetIMDB` class.

f) [DL] Read the raw `.csv`-data via Pandas and call the `prepare_data` of the previous subtask.

g) [DL] Transform the results into a `TensorDataset` and split the data twice. First, generate the training and testing data and afterwards split the testing data again to obtain the validation dataset.

h) [DL] As usual, define the three data loaders. You can verify your script by executing the `rnn_lstm.py` file (see the main function at the bottom) or printing

```python
print("training: ", len(self("train").dataset))
```

[5]The index will be important to determine the input dimensions of our classifiers.

```
print("validation: ", len(self("val").dataset))
print("testing: ", len(self("test").dataset))
```

at the end of the constructor.

2) **RNN and LSTM: Unlock the power of your past experiences.**
In this task, we will work with the file `rnn.py`.

  a) [N] Complete the `SentimentRNN` class.

   - Implement the constructor of the class. Your network should consist of an Embedding layer[6], a RNN layer[7] and a final linear layer.
     Note: It is really important to set `batch_first=True`, as our preprocessed dataset is in a column-based format.
   - Implement the forward pass of the class. Therefore, carefully read the RNN's docstring and check the dimensions of intermediate results. As usual: **the choice of your loss function determines if you need an activation function after the last layer**.

  b) [N] Complete the `SentimentLSTM` class. You can proceed analogously, just substitute the `RNN` layer with a `LSTM` layer[8]. Do not forget to use `batch_first=True`.

  c) [NT] ($\sim$ 30 min, 10 epochs) Open the `04_main.py` file, instantiate the `DatasetIMBD` class and use the `NetworkTrainer` to train your networks. Choose a reasonable loss function, an optimizer and a scheduler.

  d) [NU] Use the `NetworkUser` to evaluate your networks on the testing set. What do you observe? Which method performs better and which one takes longer to train? Does this match your expectations?

3) **BONUS:**

  a) As in the previous three exercises: Implement a method to compute the correct predictions in percentage points. Depending on how you implemented the loss function, you might need an additional activation function call.

  b) Generate some sentences by your own and observe the predictions of the classifier. Additionally, it could be informative to determine the level of certainty of the classifier's predictions by extracting the probability. **Give feedback on the programming sessions and let the classifier investigate whether you liked the workshop. Afterward send the feedback to** email.

  *c) Modify the value of the `top_n_words` parameter and analyze how the network's performances depend on the size of the vocabulary.

  *d) We have several options to optimize the performance of the preprocessing and our classification pipeline:

   Industry Level Tokenization Currently the tokenization process in the preprocessing pipeline is really slow and not scalable. *Spacy*[9] provides fast and scalable tokenization algorithms at "industry standard". Choose one tokenizer and reimplement your preprocessing pipeline.

   Classification Architecture A simple way to modify your architecture is to change the number of (hidden) features inside your `SentimentRNN` respectively `SentimentLSTM` class. Additionally, you could try to use more layers. Another interesting approach is presented in [Jou+16]. Implement the "FastText" architecture presented in the paper.

[6]See https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
[7]https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
[8]See https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html#torch.nn.LSTM
[9]See https://spacy.io/api/tokenizer

Sparse Gradients For larger corpora the encoded vectors become more and more sparse. Thus, one option is to enable the `sparse=True` flag in your network's layers and to use an optimizer that supports sparse gradients, e.g. `torch.optim.SparseAdam`.

*e) The dataset [Maa+11] is a relatively small benchmark dataset. You can run the same Sentiment Analysis to classify Amazon reviews, cf. [ZZL15] and here available to download, or take a look at the datasets provided by *TorchText*[10].

*f) Here you can find state-of-the-art methods to perform sentiment analysis in PyTorch.

*g) As you learned in the theoretical session, another commonly used network architecture is a **Gated Recurrent Unit** (GRU) RNN which can also be used for sequential input data. Read the docstring and add a GRU RNN as a third architecture to your code. To simplify your code, you could implement a `RNNFactory` class similar to the `ResNetFactory` class used in Assignment 3. How do the results differ?

References

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[Jor86] M I Jordan. "Serial order: a parallel distributed processing approach. Technical report, June 1985-March 1986". In: (May 1986). URL: https://www.osti.gov/biblio/6910294.

[Jou+16] Armand Joulin et al. "Bag of Tricks for Efficient Text Classification". In: *CoRR* abs/1607.01759 (2016). arXiv: 1607.01759. URL: http://arxiv.org/abs/1607.01759.

[Maa+11] Andrew L. Maas et al. "Learning Word Vectors for Sentiment Analysis". In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. URL: http://www.aclweb.org/anthology/P11-1015.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Ed. by David E. Rumelhart and James L. Mcclelland. Cambridge, MA: MIT Press, 1986, pp. 318–362.

[ZZL15] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. "Character-level Convolutional Networks for Text Classification". In: *CoRR* abs/1509.01626 (2015). arXiv: 1509.01626. URL: http://arxiv.org/abs/1509.01626.

[10]See https://torchtext.readthedocs.io/en/latest/ and https://pytorch.org/text/stable/index.html.