Introduction to Deep Learning

Master's in Electrical Engineering and Information Technology
Department of Electrical and Computer Engineering
Technical University of Munich
A.Y. 2021/2022

Assignment 2

Using NLP to classify celebrities' tweets

Emma Benedetti
03754006
emma.benedetti@tum.de

***Abstract*—This report will explain the procedure used to solve the second assignment of the*Introduction to Deep Learning* course, offered at the Technical University of Munich during the A.Y. 2021-2022 and taught by Dr. Hyemin Ahn. The goal of this assignment is to correctly classify tweets written by celebrities and already collected into a training and a validation dataset.**

***Keywords*—Natural Language Processing, Recurrent Neural Networks, Text Analysis**

I. Introduction

Natural Language Processing (NLP) is a powerful branch of Deep Learning that encompasses all the algorithms tasked with recognising human-based language. These algorithms are divided into two main categories, Recurrent Neural Networks (RNN) and Transformer-based methods, but the latter are outside of the scope of this project. The goal of the project is to apply different RNNs to train a model that classifies tweets written by different celebrities. In order to do that, the model is trained on a pre-processed training dataset and then evaluated on a given validation dataset, each organised as JSON files. The computations were carried out through the *PyTorch* library, and in particular its module *torch.nn*.

II. Overview of prior elements

Before describing the course of the project, it is necessary to shed light on the initial material made available for it and preliminary limitations that have been applied.

A. Available material

In order to carry out this project, documents and databases containing all essential material were made available:

1) *train.json* and *valid.json:* JSON files containing respectively the training and validation data. These files are lists containing respectively 6400 and 356 samples, organised as dictionaries with the following keys:
   - *user_id:* integer between 0 and 7 that identifies the author of the tweet;
   - *sentence:* string of the tweet;
   - *token_id:* list of tokens, each identifying a specific word of the tweet.
2) *meta.json:* JSON file containing the dataset's metadata, namely relevant information for the tweet classification. It is organised as it follows:
   - *tokens:* list of the tokens of each word;
   - *num_user:* integer reporting the total number or users (otherwise the dataset's classes)
3) *baseline.ipynb:* notebook where the data is loaded and batched, as well as the notebook where the model is defined and trained;
4) *test.ipynb:* notebook where the trained model is evaluated on *valid.json*.

B. Project Boundaries

Two fundamental limitations were applied to this project:

1) *Choice of Architecture*: it was strictly forbidden to use Transformer-based architectures. However, it was allowed to use attention-based mechanisms.
2) *Number of Parameters*: independently of the choice of model, it was not allowed to use architectures with more than 20 million parameters.

III. Procedure

In this section will be provided a description of the methodology used to find the best results in the project.

A. Data preparation

As stated before, all the model training took place in the *baseline.ipynb* notebook, which presented some precompiled code to facilitate the training. After loading the training, validation, and meta datasets, as well as a general description of each of them, the baseline asked to define the `tweetDataset` dataset and its related dataloader.
There was already an initial skeleton of the dataset, but it accepted only one token sequence per batch. Therefore, the first task was to make changes to `tweetDataset` so that it accepted batches with size larger than 1.
To do that, first it was computed the maximal length of the sequences in both the training and validation datasets (which amounted respectively to 60 and 47 tokens), then the largest between the two values was attributed to the variable

`max_len`. From then, each token list in the `tweetDataset` samples that was shorter than `max_len` was padded with a sequence of the same *End of Signal (EOS)* token until it reached that length.
This padding procedure allowed to widen the batches of sequences for the training, which was set to the default size of 32 to ensure a sufficient number of batches during the validation.
Besides that, each sequence was converted into a Pytorch Tensor for an easier application of the Pytorch module.

B. Model preparation

After preparing and loading the data in batches, a RNN model was set up with equal input parameters and embedding techniques, but different architectures that will be explained later in this section.
The model took the following variables as input:

- **num_token:** the number of tokens, collected in *meta["tokens"]* (default: $13,369$). This number did not include the *EOS* token, which was added at the moment of the Model initialization;
- **num_user:** the number of classes/labels in the classification (default: 8);
- **embed_dim:** output dimension of the embedding and consequent input of the RNN architecture (default: 512);
- **rnn_dim:** output dimension of the RNN (default: 1024);
- **num_layers:** number of RNN layers (default: 2).

For the NLP classification it was also necessary to include an embedding layer, which in this model was computed through the PyTorch API `torch.nn.Embedding`, setting as input and output dimensions *num_token* and *embed_dim*.
Afterwards, the embedded tensor was fed into the model backbone, which was set up by applying the `torch.nn` classes `RNN`, `GRU`, or `LSTM` according to the applied RNN.
Finally, there were included a Linear layer and a Dropout layer with dropout probability equal to 0.2.

C. Model architectures

The three most famous RNN architectures available on PyTorch have been trained to find the best-suited model:

1) *Recurrent Neural Network (RNN):* RNNs improve the common Neural Networks for NLP by going through every word in the sequence and memorizing key information. However, the computation of RNNs is slow and they suffer of the Vanishing Gradient Problem, namely the information held by each word becomes less and less important the "older" the word becomes in the analysis. Therefore, it might not be adequate for this analysis, but it was included to compare its results to those of more recent architectures.
2) *Long Short-Term Memory (LSTM):* LSTM models were created to solve the above described Vanishing Gradient Problem in RNNs. Rather than retaining information from previous words, LSTM will estimate the importance of the word in the sequence.
3) *Gated Recurrent Unit (GRU):* Gated Recurrent Units are similar to LSTM networks, but unlike them GRUs store each word's information into their hidden state and pass it onto the next GRU.

One singular characteristic of the RNN PyTorch implementation is that there is the possibility to decide which non-linearity function, between `tanh` and `ReLU`, should be applied in the network. However, since the two other architectures defaulted to `tanh`, this non-linearity function was chosen also for RNN.

D. Training

As for the training, some common parameters were used independently of the backbone of the model:

- as **optimizer** was used the *Adam* algorithm, with learning rate equal to $1e-03$ and the weight decay parameter set to $1e-09$;
- **Cross-Entropy** was chosen as loss function;
- As quality measure, it was used the **accuracy** of the model.
- Each model was trained on **100 epochs**;

An attempt was also made to add a learning rate scheduler, but this was impossible to implement as the accuracy dropped dramatically during its use.
There has also been an attempt to train the models on a number of layers higher than 2. However, while the number of parameters clearly increased, the accuracy dropped drastically and it was not deemed necessary to report it. Therefore, only the results of the architectures trained on 1 or 2 layers are reported in the table in the `Results` section.

IV. Results

The table below summarises the most important results of each architecture: It can be seen that the performance of a standard RNN is much worse than the performance of the two other information-retaining architectures. This suggests that the Vanishing Gradient was a significant obstacle for the classification, and therefore made the model inadequate for this analysis.
However, another important result to notice is the similarity between the results of a 1-Layer LSTM and a 2-Layer GRU, which suggests that the information retained in each model might be almost the same. However, adding a second layer to the LSTM resulted in a lower accuracy. From this information it can be hypothesized that, unless the LSTM is not boosted

TABLE I
Results obtained by each trained model

Architecture	num_layer	Validation accuracy
RNN	2 Layers	14.04%
GRU	2 Layers	76.68%
LSTM	1 Layer	75.84%
	2 Layers	55.90%

with an Attention layer, it actually benefits from the possible lowest level of layers.
Finally, since the GRU resulted in a higher accuracy, it has been chosen as the model to submit as final result of this project.

V. CONCLUSION

The goal of this project was to apply NLP techniques to assign the tweets from a pre-processed database to their rightful author. To do that, the three most common Recurrent Neural Network architectures (namely RNN, LSTM, and GRU) have been trained and evaluated, which resulted in different levels of accuracy. However, the highest-quality method resulted in an accuracy of about 77%, which is much lower than State-of-the-Art, Transformer-based models. However, this accuracy can also be attributed to the absence of more elaborate embedding and memory-retaining methods inside of the trained arcitectures.