UWr, Machine Learning 2019/2020

Final project by *Wojciech Pędrak*

Whats, wheres and hows

What is the purpose?

The goal of the project was to get familiar with neural networks (referred later as NN). To archive that the following actions were taken:

1. Data:
   - We've selected dataset to work on.
   - We've cleared the data and prepared to process
2. Non-NN models:
   - We've chosen a few baseline models using `sklearn` with default values
   - We've tuned parameters for chosen models to make it as effective as possible
3. NN models:
   - We've proposed two simple baseline models
   - We've proposed upgrades to the selected models
   - We've tuned hyperparameters of one deep neural network to find out the best setup

Where to find?

Project can be found on github or colab.

How to read?

Most of the important sections have a 2-3 sentence introduction to help understand the following code. In this doc, we will rather focus on high-level understanding and learnings rather than code analysis.

Data:

Dataset can be found on kaggle.
in the dataset, there are 40.000 tweets. Each of them is labeled by one of 13 sentiments. Distribution of classes is uneven (refer to the plot).

In the scope of cleaning the data, we lowered all the tweets, remover all special characters and lemmatized each word. Further, we represented each tweet in two ways:

- Bag-of-words [wiki]
- TFIDF [wiki]

Models (non-NN):

The first thing to mention is that for comparison of models we are using accuracy, which probably is not a good choice in the real world, but we will use it for simplicity.

We've selected 4 models out of the `sklearn` package and used them with (almost) default parameters.

1. `MultinomialNB`
2. `LinearSVC`
3. `RandomForestClassifier` - lowered number of trees to make it faster
4. `LogisticRegression` - increased max number of iterations to get rid of warnings

The best score so far was `0.344` by `LogisticRegression`.
After that, we tuned the parameters of each model to make it better than the default one.

	default	tuned param	after tuning
MultinomialNB	0.321	alpha	0.326
LinearSVC	0.328	C	0.351
RandomForestClassifier	0.304	max_depth, min_samples_split	**0.355**
LogisticRegression	0.344	C	0.348

Values in the table are max of scores on Bag-of-words and tf-idf representation. It gives us `0.355` accuracy baseline that we will try to beat using neural networks.

Models (NN):

For the purpose of simplicity, from this point, we will use only the tf-idf representation of data.

Baseline

We start with two simple models: tiny and big. it happened that smaller model yields (slightly) better performance (`0.334` vs `0.330`). The difference is quite small, but the reasons are actually interesting. It happened that the bigger model overfits data after the first epoch where tiny one scored the best result after 5 epochs.

Dropout

To make the process of overfitting slower and (hopefully) get better accuracy we've introduced the `Dropout` layer in both of our baseline models. It is setting a fraction of its inputs to 0 at random. To help with intuition: the next layer after Dropout sees only randomly selected fraction of data that was passed to the dropout layer.

Scores with an additional layer on the tiny model did not change (`0.336`) however, the model reached max accuracy after 11 epochs. It makes sense. The tiny model didn't have a problem with overfitting, so "not seeing " all the data only slowed down the whole process of learning.

We've got a little boost on the big model. It yields `0.344` accuracy.

Word Embeddings

Here we've changed the representation of tweets. Now we will embed words into space where similar words are close to each other. Intuition is that if similar words are close, then similar emotions should somehow be also close.
First, we've learned embedding ourselves and then we used pre-trained one.

Learning embedding ourselves gave us `0.322` on tiny and `0.356` on big. It looks like the tiny model is out of learning power and big one started to outperform it.

Let's check pre-trained embeddings. We've used two of them: `glove.6B` which was learned on Wikipedia and `glove.twitter.27B` learned on tweets. Both embeddings are available thanks to GloVe and can be found online:

- glove6B (~800MB)
- glove.twitter.27B (~1.4GB)

Results of the tiny model on:

- glove6B: `0.257`
- glove.twitter.27B: `0.318`

Results of the big model on:

- glove6B: `0.355`
- glove.twitter.27B: `0.357`

So it looks that pre-trained embedding is not helping at all. Why it is like that?
Probably because of low coverage of our vocabulary (set of words that occur in any tweet) by pre-trained embedding. Example: there is no embedding for the word "soooooo".

Convolutional layer

Next, we've added the `Convolutional` layer that is said to works well for identifying simple patterns. We will add it after the embedding layer. This time we will use only twitter embedding as it was performing better in the previous section (at least for tiny model). In fact, there is no gain from it. The tiny model scored `0.279` and the big one scored `0.359`.

Tune hyperparameters

Let's remind all the previous results:

	pure	**dropout**	**word embedding (self)**	**word embedding (external)**	**convolutional**
tiny	0.334	0.336	0.322	0.318	0.279
big	0.330	0.344	0.356	0.357	**0.359**

So far we've got the best score of `0.359` which is better than any of the baseline scores however, it is not that big difference. One more thing that we've done is (as on baseline models) tuning of parameters. We've created a model that consists of all proposed layers and then runs it for many params to find the best combination. By doing so, we've got **`0.384`** as the final result.

Summary

Conclusions

The first take away lesson is that tuning parameters for models is the hardest and most time-consuming part of machine learning. On this data, random forest after tuning was almost as good as a complicated neural network.
The second lesson is that writing neural networks is easy nowadays. The best model in this project takes ~10 lines of code. It doesn't mean that it is easy to write well-performing neural network though.
Third and the last lesson is that time matters. We've resigned from using the `SVC` classifier (kernelized SVM) as it took too long to fit it even on a chunk of data. Neural networks aren't fast either, but there are easily (from point of view of the user) parallelizable. It's two-liner to make model work in parallel on GPU and 3 clicks to get runtime with GPU from colab.

What wasn't done?

- [ ] More data cleaning
- [ ] Taking into account that the distribution of labels is uneven
- [ ] Normalasing data to get higher dictionary coverage (in embedding methods)
- [ ] Using different function than accuracy for success measurement