Learning from Data – Assignment 3: Recurrent Neural Networks & Pretrained Language Models

General remarks

This assignment has to be completed in **groups of 3**. This assignment uses the data set from week 1 again (so not the one from previous week!). We will work only with the multi-class classification problem (so not sentiment analysis). You are provided with an almost-working implementation of an RNN and are asked to make modifications. Ultimately, you will employ pretrained language models to push performance.

What you have to hand in:

- A report based on the template (see week 1) that addresses all questions raised in the assignment. Remember to structure your report as an academic paper! You also have to submit your code this week.
- A paragraph (either in the report or on Brightspace) explaining who did what exactly.

Deadline: Monday, October 16th, 10.59

Data and Software

For this assignment, we will be using the data set from week 1 again. Again, we will only be focusing on the **6-class text classification problem**.

The assignment files also contain pretrained Glove embeddings (of embedding size 300). For efficiency, they are filtered to only contain the words present in the train, dev and test set. Note that this assumes you do no touch the existing tokenization. If you do so anyway, you might want to download your own embedding file.

For this assignment, we will be working with Keras again. Note that as these models keep getting bigger, you might want to use a GPU. See later below for details on how to use one using Google Colab.

3.1 – Classification using LSTMs

You are given the file `lfd_assignment3.py`. If you try to run it, you will notice it does not work yet! Currently, it reads in the pretained word embeddings and uses that as an initial embedding layer. It also already has an output layer for making predictions. Your task is to add the layers that actually implement the recurrent neural networks. Start out by using an LSTM layer in Keras:

`https://keras.io/api/layers/recurrent_layers/lstm/`

You will also see that this script implements **early stopping**. We specify a max number of epochs, but stop training if the validation loss does not improve for three consecutive epochs. You do not necessarily have to keep this in, but I do recommend it, as it helps in comparing models across different settings and architectures.

Also note that the TextVectorization step, the we actually cut-off the input after 50 tokens! You can experiment with this number, but it was mainly added to make sure your models can still run reasonably well on a CPU.

Running on a GPU

Still, the run times might be too slow for your wishes. You can speed up the training times a lot by using a **GPU**. You are free to use any GPU setup you want, but here I'll show you briefly how to work with Google Colab:

`https://colab.research.google.com/`

On Colab, just start a new notebook. First, we have to make sure we have access to all the files. The easiest thing is probably to upload all files (data, embeddings, Python script) to your Google Drive, in a folder called AS3. You can now mount this folder by running the following command in your notebook:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

This will give you a link you have to follow to grant access by using a code. The files are now available in `/content/gdrive/MyDrive/AS3/`. Now, make sure we are actually using a GPU:

- In the top menu, go to Runtime → Change runtime type → Hardware accelerator: GPU

From here, you have two options. You either convert the Python script to a notebook of your liking (i.e. first loading all the data and then easily re-running certain experiments). Or you can just run the Python script from here, since we put it on our Drive. If you change things, you just have to re-upload the script to the Drive. I'll show how to use the latter:

```
!python /content/gdrive/MyDrive/AS3/lfd_assignment3.py --train_file
/content/gdrive/MyDrive/AS3/train.txt --dev_file /content/gdrive/MyDrive/AS3/dev.txt
--embeddings /content/gdrive/MyDrive/AS3/glove_reviews.json
```

This should automatically run on a GPU, no change in code required. It will be a lot faster!

Experiments

Remember that you always have to tune neural networks. Make some changes to the learning rate, batch size, etc to get an LSTM model that does reasonably well in terms of accuracy. You can treat this as your "baseline" model for the next experiments. Of course, all these experiments should be performed on the development set and not the test set! For the assignment, I want you to look at a number of settings/architectures, and I want you to report whether you could get any improvements in accuracy. Be precise!

- Experiment with the effect of pretrained embeddings. The current model adds them, but you can also leave them out. Moreover, you can put "trainable" to true instead of false (what does this mean anyway?) Or add an extra dense layer between the embedding layer and the LSTM.
- Experiment with extra LSTM layers - do they help? How do you need to modify the LSTM call to be able to stack multiple layers? And why?
- Experiment with dropout (check the LSTM function arguments). Is there a difference between dropout and recurrent dropout?
- Experiment with optimizers: is SGD the best option? Remember that you likely have to change the learning rate as well.
- Change the LSTM to a bidirectional model. Do the scores improve?
- Of course you are free (and encouraged!) to do any other interesting experiments you can think of!

Given these experiments, try to combine certain improvements and select your best model. You might to re-visit certain settings such as the batch size and the learning rate: optimal settings could be different now!

Use the test set to get an accuracy score of your best LSTM model. Since you are doing lots of experiments, you might have overfit on the dev set. Therefore, make sure some of the improvements you found still hold up on the test set (i.e. have test set scores for 3-5 models that had worse dev set performance). Discuss the results.

3.2 – Pretrained Language Models

We will now try to push performance by using pretrained language models (LMs). You will need to modify the code in the Python script to be able to work with these new models. You are free to use any implementation you want, though to keep it somewhat compatible with the current script I'll give examples using Tensorflow.

Setting up

Even if you managed to do the previous exercise without a GPU, you'll probably want to use a GPU now. I recommend setting up a notebook and following along with the code here. You will also need to copy functionality from lfd_assignment3.py: the data loading and label

binarization mainly. In the next instruction I expect you to have loaded the data.

First, we need to install the `transformers` library, developed by HuggingFace:

```
pip install transformers
```

Let's start the with the simplest route: directly import the model we can use for classification:

```
from transformers import TFAutoModelForSequenceClassification
```

This is nothing more than a linear classification layer on top of the LM model outputs. We only have to specify the string identifier (say "bert-base-uncased") to be able to use this particular LM automatically. Isn't that great!

A lot of the work is therefore being done behind the scenes, so please check out how it works here (scroll down for specific examples):

```
https://huggingface.co/transformers/model_doc/auto.html
```

Remember that LMs usually have their own particular type of tokenization (BPE), so we have to make sure we tokenize our input the correct way. Wouldn't you know it, there also exist automatic tokenizer for the corresponding LMs:

```
from transformers import AutoTokenizer
```

Again, it's great that this functionality exists, but it also obfuscates what is going on exactly. Please check out some of the documentation of the AutoTokenizers.

There's no need for the pretrained embeddings anymore, as we will be using a full pretrained model. Let's use the **BERT-base** model first. We read in the data as we did before (including binarizing the labels), which you should copy in the notebook if you use Google Colab. Then, we use the tokenizer to preprocess the data like this:

```
lm = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(lm)
model = TFAutoModelForSequenceClassification.from_pretrained(lm, num_labels=6)
tokens_train = tokenizer(X_train, padding=True, max_length=100,
               truncation=True, return_tensors="np").data
tokens_dev = tokenizer(X_dev, padding=True, max_length=100,
             truncation=True, return_tensors="np").data
```

Since we use byte-pair encoded input I've set a max_length of 100 instead of 50, but you can play around with this. Also note that we pad the input if the length does not reach 100.

Now let's finetune this model! Really, these models can be tricky to train, especially regarding learning rate and optimizer. I'll give you a couple of settings that work well, so you do not waste too much time here. First, it's important to use `from_logits=True` in the loss function, as the classifier on top of BERT does not have a softmax:

```
from tensorflow.keras.losses import CategoricalCrossentropy
loss_function = CategoricalCrossentropy(from_logits=True)
```

Secondly, it's usually better to use the Adam optimizer, with a smaller learning rate than the default:

```
from tensorflow.keras.optimizers import Adam
optim = Adam(learning_rate=5e-5)
```

Now we can simply compile and train our model as we did before.

```
model.compile(loss=loss_function, optimizer=optim, metrics=['accuracy'])
model.fit(tokens_train, Y_train_bin, verbose=1, epochs=1,
          batch_size=8, validation_data=(tokens_dev, Y_dev_bin))
```

Similarly, we can make predictions with model.predict() as well (though we have to get the logits now):

```
Y_pred = model.predict(tokens_dev)["logits"]
```

Simply running this for 1 epoch should give you an accuracy of 93%! If you get a lot lower, something went wrong somewhere.

Assignment

For the assignment, I want you to experiment with the different pretrained language models and the settings they have. This includes:

- Improving the performance of a single language model. You can experiment with the max sequence length, learning rate, batch size and number of epochs. You can also look at using a learning rate scheduler (e.g. PolynomialDecay, google it!), for example. Do a final run on the test set to check if you really improved. Discuss the results.
- Experiment with using different pretrained language models. There are many of them! Select a number of them that you find promising (and motivate why) and compare results. Is there anything that stands out? What model gave you the best performance? Is there a large difference between them? Maybe a more efficient LM is worth a small drop in performance.

For the report, remember to be precise, especially regarding the settings you used to obtain certain results. Make sure that for each table in the report it's clear to the reader how you obtained the results. Remember that in lecture 2, I gave a lot of advice regarding the structure of the report.

For this report in particular, it's nice if you give a small explanation of how LSTMs work. Also, it's good to include a brief section on how the LM classification works (and how LMs are trained). If you use multiple LMs, it's good to highlight how they are different. Please cite the corresponding papers of the LMs that you used in the report. And of course, there's still room for error analysis, or doing an extra analysis on the performance of a certain class (e.g. also reporting F-scores per class, confusion matrix, etc).

Remember that on Brightspace there's a long list of resources regarding LSTMs, Transformers and pretrained language models! Good luck!