MT Exercise 2

Topic: RNNs and Language Modelling

Due date: Tuesday, 1 of April 2025, 14:00

Submission Instructions:

- **Submission file format: zip**
- Please follow our file naming convention: `olatusername_mt_exercise_xx.zip`, for instance `mmuster_mt_exercise_03.zip`
- **Submit URLs to Github forks:** Part of your work should be committed to your Github repositories. Your submission to OLAT should therefore include the links to those repositories (e.g. in the PDF document with the results from `task 1`).
- Please submit via the exercise submodule on OLAT. Submission is open only until Tuesday, 14:00.
- You are strongly encouranged to work in groups of two. If you submit as a group, please include both usernames in the submission.

Alternative Instructions:

If you would like to avoid creating a Github account, or making your work available in a public repository, here are some alternative submission instructions.

- Do not create a Github account. Instead of forking repositories to your account (ignore the instructions in the exercise below that tell you to do this), simply clone our repositories to your local machine.
- Commit your changes locally, without pushing them to a remote repository.
- **Submit ZIP folders of repository files:** For each of your Github repositories, create a ZIP folder with all files. Go to the local copy of the repo on your machine, then zip everything, including a very important hidden folder called `.git`. On a Unix machine, for example:

```
cd [your local copy of the repository]
zip -r repo.zip .
```

1 Training a recurrent neural network language model

In this exercise you will train a language model using a recurrent neural network (RNN). As the backend library, we will again use Pytorch (same as for JoeyNMT). Pytorch includes a repository with examples, one of them is word-level language modeling:

`https://github.com/pytorch/examples/tree/master/word_language_model`

This is the code we will use as a starting point for our language model. After setting up, your task will be to decide on an interesting data set, prepare this new data set and train a new model.

Setting up

For convenience, we prepared a repository for you that showcases all the commands that lead to a trained language model[1]:

`https://github.com/marpng/mt-exercise-02`

Important: Before cloning this repository, fork it to your own Github account[2]. This allows you to make changes to this repository and commit them. After forking, clone your forked repository in the desired place on your machine:

```
git clone https://github.com/[your username]/mt-exercise-02
cd mt-exercise-02
```

Then create a new virtualenv, and activate it:

```
./scripts/make_virtualenv.sh
source venvs/torch3/bin/activate
```

Then install packages and clone all the required code:

```
./scripts/install_packages.sh
```

This will install Pytorch, sacremoses and also clone the Pytorch examples repository. Even though we prepared all of those scripts for you, please study them closely and understand all individual steps.

Getting to know the required data format

The repository contains an example data set, now on your machine here:

```
ls tools/pytorch-examples/word_language_model/data/wikitext-2
# README  test.txt  train.txt  valid.txt
```

That you could use for training right away. Looking more closely at this example data set you will notice that it is split up into a training, validation and testing part, and that the training part is about ten times larger than validation and test data:

[1]A big thank you goes to Mathias Müller, the original author of these scripts.
[2]`https://help.github.com/en/github/getting-started-with-github/fork-a-repo`

```
wc -l tools/pytorch-examples/word_language_model/data/wikitext-2/[tv]*
#  4358 tools/pytorch-examples/word_language_model/data/wikitext-2/test.txt
# 36718 tools/pytorch-examples/word_language_model/data/wikitext-2/train.txt
#  3760 tools/pytorch-examples/word_language_model/data/wikitext-2/valid.txt
```

Also, looking at some actual lines of text, you will notice that some preprocessing was already applied to this data. Here is a random line from the training set:

```
<unk> packages of ordnance and ordnance stores received and
mostly issued to troops in service .
```

This will tell you that the data is tokenized already, but also that some tokens have been replaced with a placeholder token for an unknown word, <`unk`>. Replacing some of the words with <`unk`> is done to create a vocabulary (set of known, unique words) with a fixed size. Usually, words are sorted by their frequency in the corpus, and then a cut-off point, such as `10000`, is defined. A vocabulary size of `10000` would mean that the top `10000` most frequent words are left untouched, while all others are replaced with <`unk`>.

Vocabulary size has an impact on model size, since it determines the number of embedding parameters that need to be learned. Having a smaller vocabulary size will mean that your model has a smaller memory footprint, but also that some words will be unknown to the model, and cannot contribute to learning.

For a different data set, there are important consequences:

- The Pytorch language model code expects data to be preprocessed in exactly the same way, meaning: the data needs to be tokenized already, and if you would like to reduce the vocabulary size, you have to do so in preprocessing as well.
- The code also assumes that you split up your data into a training, validation and testing part and call them `train.txt`, `valid.txt` and `test.txt`.

Finding an interesting data set

Keeping in mind the required data format, your task is to find an interesting data set on your own, train a model and eventually generate some new text similar to the training text.
Additional things to consider regarding the choice of data set:

- Data sets should have a certain size. As a crude rule of thumb, your preprocessed data set should contain between `5000` and `10000` segments.[3] For a data set of roughly this size, a vocabulary size of 5000 makes sense.
- It does not need to be an existing data set. You can also create your own, synthetic data set for interesting experiments. To give an example: how about a synthetic data set consisting of different movie dialogues?

[3]More segments tend to lead to better and more interesting results when generating text, but also affect the training time of the model.

Training with your own data

Once you have an interesting data set, preprocess it as needed. As an example, our repository shows how to obtain a collection of Brothers Grimm fairy tales from Project Gutenberg, and how this particular data set would need to be preprocessed:

```
./scripts/download_data.sh
```

Again, please appreciate that we prepared all the necessary code for you and study those scripts in detail. After preparing custom data, you can start training a model. For the Grimm data set, the training can be invoked as follows:

```
./scripts/train.sh
```

This script calls the Python script `main.py` from the Pytorch examples repository. On a typical laptop machine on CPU, a training using the default parameters in `main.py` should take between 20 and 40 minutes (depending on the size of your data set). If you have a GPU machine, more or fewer CPU cores than 4, or a MacOS GPU, you need to make some changes to this script for it to run with optimal efficiency. This will depend or your hardware but the time savings are well worth it. Training a model on GPU can take as little as 90 seconds for example.

Training hyperparameters

Training a language model has a number of important hyperparameters, for which you, the user, have to define values. For instance, our example training script says to train for 40 epochs, that some statistics on the training set should be reported after 100 batches (once per epoch for the example data set), that the embedding size is 200 and that dropout should be 0.5. Those are more or less arbitrary settings: for each data set, different hyperparameters will be ideal.

It is not always easy or possible to have an intuition how to set hyperparameters. So, what we do very often in practice is train several models, with different hyperparameter settings, to see which one works best. For language models, “working well” usually means having a low **perplexity** on a test set.

Our Pytorch training code computes three different perplexities:

- training perplexity after each epoch

  ```
  | epoch  40 | [...] | loss  4.30 | ppl    73.56  # ppl = perplexity
  ```

- validation perplexity on the valiation set after each epoch

  ```
  | end of epoch  40 | [...] | valid loss  4.47 | valid ppl    87.74
  ```

- test perplexity on the test set as the very last step in training

```
| End of training | test loss  4.29 | test ppl    72.77
```

Apart from having a low perplexity, other important considerations are of course: how much time and memory it takes to train a model. In general, larger models work better, but take longer to train. For this exercise, it is impractical if training takes very long. Rule of thumb: if training takes longer than 2 hours, please change some settings.

Generating some text from your own model

After training the model, you can use it to sample a snippet of new text that will be similar to the training material. As an example, run

```
./scripts/generate.sh
```

and the sampled text will be saved in the file `samples/sample`. There are hyperparameters for sampling as well, such as temperature (how “creative” the model should be), or length of text to generate.

Your task: training your own language model

Find or create an interesting data set that meets our requirements, and preprocess this new data set, for instance by changing the script `scripts/download_data.sh`. Afterwards train a model with the standard settings and generate some sample text.
Your submission should discuss the following points:

- Present your chosen data, does it have any special attributes that you expect to have an influence on the text generation of the model?
- Take a look at the sample generation, what are your impressions?
- Modify the README.md in your repository to document the changes you made.

Submission: PDF document with your findings and a link to your forked repository.

2 Parameter tuning: Experimenting with dropout

Dropout has become an essential hyperparameter for neural networks and is also a standard hyperparameter in machine translation. However, the choice of the right dropout setting is not an easy one. One approach to find the best setting is to compare the perplexities of different models using varying dropout settings.

For this exercise,

- Train at least 5 language models with varying dropout settings (including a model without any dropout), for instance by changing the script `scripts/train.sh`. We recommend that you choose a value between 200 and 300 for the embedding size and the number of hidden units per layer. Train on the data you prepared in Task 1.
- Create for each of the three different perplexities a table with the values of each of the different models per epoch/end of training. Each of the tables should look something like this (but with more models and epochs):

Valid. perplexity	**Dropout 0**	**Dropout 0.3**	**Dropout 0.6**	**...**
Epoch 1	94.32	96.33	98.41	
Epoch 2	87.79	92.79	95.81	
Epoch 3	73.21	86.26	88.58	
Epoch 4	68.35	82.51	85.37	
Epoch 5	62.97	77.80	81.78	
Epoch ...				

  To achieve this, modify `/tools/pytorch-examples/word_language_model/main.py` so it accepts and additional flag to save the perplexities as a log-file. We recommend you use a tabular format for this purpose, which will make further processing easier. You are allowed to import additional packages. Commit a copy of `main.py` to your repository.
- Create a line chart each for the training and the validation perplexity to visualize the results. Write and commit a python script for creating the tables and line plots, taking the log files from the previous step as input.
- Can you see a connection between the training, validation and test perplexity? Based on your results, which dropout setting do you think is the best and why?
- Sample some text from the model that obtains the lowest test perplexity, for instance by changing the script `scripts/generate.sh`. What do you think of its quality? Does it resemble the original training data?
- Sample some text with the highest test perplexity. Can you see a difference to the lowest scoring one?
- Describe your changes, the commands to run and in what order in the README of your forked repository.

- **Important, read very carefully: Document all of your changes and additions by committing them to your forked repository. This means that we should be able to clone your forked repository, and there should be instructions for us to download your data, preprocess and train your models. We will not consider commits that are made to this repository after the submission deadline.**

Submission: A PDF with your findings, A link to your forked repository, as well as the line-plots and the tables.

In case of problems or if exercises are unclear please post in our OLAT forum.
Good luck!