Practical set-up & Training data

This page will contains materials for the first tutorial session (April 19th).

The learning goals for the first tutorial are:

- preparing the Python requirements for practical exercises in the upcoming tutorials,
- test-running a few lines of code,
- familiarization with a few coding best practices,
- understanding key processing steps and terms of the first building block for training any language model -- the training data.

Please try to complete the first block of this tutorial sheet (i.e., installation of requirements) AHEAD of the tutorial session, ideally, while you have a stable internet connection. This way we can try to solve any problems that might have come up with the installation during the tutorial on Friday.

Installing requirements

Throughout the semester, we will use Python, PyTorch and various packages for practical work. Both the in-tutorial exercise sheets and homework will require you to execute Python code yourself. Please follow the steps below to set up the requirements (i.e., most packages required for completing exercises) that we will use in the course. We will most likely install more packages as we go during the semester, though.

You can do so either on your own machine, or by using Google Colab. You can easily access the latter option by pressing the Colab icon at the top of the webook's page. Depending on your choice, please follow the respective requirement installation steps below.

Colab

The advantage of using Colab is that you don't need to install software on your own machine; i.e., it is a safer option if you are not very comfortable with using Python on your own machine. Colab is a platform provided by Google for free, and it also provides limited access to GPU computation (which will be useful fpor working with actual language models). Using it only requires a Google account.

For using a GPU on Colab, before executing your code, navigate to Runtime > Change runtime type > GPU > Save. Please note that the provided Colab computational resources are free, so please be mindful when using them. Further, Colab monitors GPU usage, so if it is used a lot very frequently, the user might not be able to access GPU run times for a while.

Colab already provides Python as well as a number of basic packages. If you choose to use it, you will only need to install the more specific packages. Note that you will have to so *every time* you open a new Colab runtime. To test that you can access requirements for the class, please open this notebook in Colab (see above), uncomment and run the following line:

```
# !pip install datasets langchain torchrl llama-index bertviz
wikipedia

!pip uninstall torch -y

!pip install torch==2.2.1
```

Local installation

Using your computer for local execution of all practical exercises might be a more advanced option. If you do so, we strongly encourage you to create an environment (e.g., with Conda) before installing any packages. Furthermore, ideally, check if you have a GPU suitable for deep learning because using a GPU will significantly speed up the work with language models. You can do so by checking your computer specs and finding out whether your GPU works with CUDA, MPS or ROCm. If you don't have a suitable GPU, you can use Colab for tasks that require GPU access. Finally, please note that we will download some pretrained models and some datasets which will occupy some of your local storage.

If you choose to use your own machine, please do the following steps:

- install Python >= 3.9
- create an environment (optional but recommended)
- download the requirements file here
- if you have a deep learning supporting GPU:
  - please check here which PyTorch version you need in order to use the GPU
  - please modify the first line of the requirements file to reflect the PyTorch version suitable for your machine (if needed)
  - please install the requirements from the requirements file (e.g., run: `pip install -r requirements.txt` once pip is available in your environment; adjust path to file if needed)
- if you do NOT have a deep learning supporting GPU:
  - please install the requirements from the requirements file (e.g., run: `pip install -r requirements.txt` once pip is available in your environment; adjust path to file if needed)

Verifying requirement installation

Please run the following code cells to make sure that the key requirements were installed successfully. If you errors occur and you cannot solve them ahead of the tutorial, please don't be shy and let us know in the first tutorial!

```
# import packages

import torch
from transformers import AutoTokenizer
from langchain.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper
```

```
# check available computation device
# if you have a local GPU or if you are using a GPU on Colab, the
following code should return "CUDA"
# if you are on Mac and have an > M1 chip, the following code should
return "MPS"
# otherwise, it should return "CPU"

if torch.cuda.is_available():
    device = torch.device("cuda")
    print(f"Device: {device}")
elif torch.backends.mps.is_available():
    device = torch.device("mps")
    print(f"Device: {device}")
else:
    device = torch.device("cpu")
    print(f"Device: {device}")

# test PyTorch

# randomly initialize a tensor of shape (5, 3)
x = torch.rand(5, 3).to(device)
print(x)
print("Shape of tensor x:", x.shape)
print("Device of tensor x:", x.device)

# initialize a tensor of shape (5, 3) with ones
y = torch.ones(5, 3).to(device)
print(y)

# multiply x and y
z = x * y
print(z)

# testing LangChain

# run a Wikipedia query, searching for the article "Attention is all
you need"
# NB: requires an internet connection
wikipedia = WikipediaQueryRun(api_wrapper=WikipediaAPIWrapper())
wikipedia.run("Attention is all you need")

# testing the package transformers which provides pre-trained language
models
# and excellent infrastructure around them

# download (if not available yet) and load GPT-2 tokenizer
tokenizer_gpt2 = AutoTokenizer.from_pretrained("gpt2")
text = "Attention is all you need"
# tokenize the text (i.e., convert the string into a tensor of token
IDs)
input_ids = tokenizer_gpt2(
```

```
    text,
    return_tensors="pt",
).to(device)

print("Input IDs:", input_ids)
```

Best practices for writing code

There is a lot of debate around best practices for writing, documenting and formatting Python code and their actual implementation in daily practice, and many people have different personal preferences. We are not committing to a particular side in this debate, but we do care about a few general aspects:

- working with clean code
- working with understandable code (i.e., commented, with understandable variable names etc)
- producing well-documented projects (e.g., supplied with relevant READMEs etc). Think: your work should be structured such that you could look at it in a year and be able to immediately what you did, how and why.

There are a few de facto standard *formatting* practices that help to keep Python code crisp and clean. Please take a look at these and adhere to these as much as you can (as so will we):

- PEP8: style guide for Python code defining e.g., variable naming conventions, how many spaces to use for indentation, how long single lines should be etc.
  - Here is an overview video of some of the PEP8 conventions
  - There is handy software that reformats your code for you according to some of these conventions. Such software is often seamlessly integrated in IDEs. This includes for instance *Black* or *Ruff* Python formatters. They can be installed as extensions in, e.g., Visual Studio Code.
- *docstrings* are comments (strings) that document a specific code object and always directly follow the definition of the object (e.g., directly after `def fct(...)`). They specify the functionality, inputs, outputs and their types. Again, there are slightly different formatting styles for docstrings; please try to be consistent about your formatting.
  - One example style of docstrings is *numpydoc*; you might see that the provided code might often use such docstrings.

```
# example: bad formatting
def add(a,b):
    return a+b

# example: better formatting
def add(a, b):
    return a + b

# example: bad docstring

def add(a, b):
```

```
    """a+b"""
    return a + b

# example: better docstring
def add(a, b):
    """
    Add two numbers.

    Args
    ----
    a: int
        First number.
    b: int
        Second number.

    Returns
    -------
    int: Sum of a and b.
    """
    return a + b
```

There are also some hints regarding structuring larger projects and e.g. GitHub repositories (just fyi):

- project structure
- writing good READMEs
- tidy collaboration and git

These best practices will be useful to you beyond this class and possibly even beyond your studies when collaborating on other coding projects within teams or even by yourself. We do our best to stick to these guidelines ourselves and kindly urge you to do the same when submitting assignments and possibly projects.

Understanding training data

One of the critical building blocks of any language model (be it an n-gram model or GPT-4) is the **training data**. The contents of the training data determine, for instance, which tokens (e.g., words) the model "sees" during training, how often each of them occurs, which language the model learns, but also which potential *biases* the model might inherit (more on this in lecture 9).

The goals of this part of the sheet are:

- introduce core terms and concepts that we might be using throughout the class, and that are often used in NLP papers
- understand core data processing steps used before training LMs
- try hands-on loading a dataset and performing basic preprocessing steps

Tasks:

- read the sections below, try to understand each concept and ask yourself whether you have already heard it, and if so, in which context
- complete the exercises
- complete the coding exercises where you can load and process a dataset yourself.

Core concepts

- **(training) data / dataset** (in the context of LMs): a collection of text data which is used as input to the LM in order to optimize its parameters, so that, ideally, the model learns to perform well on its target task; that is, to predict fluent text. Anything from a single sentence to a book can be considered data; but since learning statistics of natural language is very difficult, usually very large collections of texts (i.e., very large datasets) are used to train LMs. Generalization to other machine learning models: the type of input data might be different (e.g., images and labels for image classification models) but the purpose is the same. Data and dataset are mostly used interchangeably.
  - **corpus** [ling.]: "A corpus is a collection of pieces of language text in electronic form, selected according to external criteria to represent, as far as possible, a language or language variety as a source of data for linguistic research." source For the purposes of NLP, the term corpus is often used interchangeably with the term dataset, especially when referring to collections of literary texts (e.g., the Books corpus) or when sourced from corpora created in linguistics.
    - well-known linguistic corpora are, e.g.: the Brown corpus, the British National Corpus BNC.
  - **test / validation data** (general ML concept): the full dataset is sually split into the *training data* (used to optimize the model), and the held-out *validation data* and *test data* (called dataset splits). Validation data is often used to optimize aspects of the model architecture (so-called hyperparameters like optimizer, drop out rate etc). This split is sometimes ommited if no hyperparameter tuning is done. Test data is then used to assess the model's performance on *unseen* data. That is, it is used to approximately answer the question: How well will my trained model perform on completely new inputs? In the context of LMs, all dataset splits are texts.
- **cleaning & preprocessing**: this is the step when "raw" data (e.g., from the web) is processed so as to massage the data into a format that is optimal for the NLP task which we want to accomplish. This can include, for instance, removing markup tags, lower-casing data, splitting it into single sentences etc.
- **annotation**: this step refers to enriching "raw" data with additional information like judgements about the quality of data, "gold standard" demonstrations of a task (e.g., gold standard answer to a question) etc, usually provided by humans. This is done generate high-quality training datasets which cannot be obtained otherwise.
  - most prominently, human annotation is often used in the process of fine-tuning LLMs with RLHF (more on this in lecture 5).
- **token**: minimal unit of text which is mapped onto a numerical representation to be readable for the LM. Different types of tokens have been used: single words, single characters, and recently mostly sub-word parts. Note that unique minimal units are assigned different tokens; whenever such a unit occurs in a particular context, the same numerical representation (i.e., token ID) is assigned to that unit. Therefore, the notion of

a token in NLP is not completely equivalent to the notion in lingusitics (and there are no types in NLP as opposed to linguistics).
    - tokenization is the process of converting a string to a list or tensor of tokens.
    - part of tokenization for training transformers is also creating *attention masks* which "mask" certain tokens for the model (i.e., hide it from the model during training). This is done to train models to predict next words based only on preceding context.
    - tokenization will be discussed in more detail in the session of week 3.
- **vocabulary**: the set of unique tokens used by a particular LM-tookenizer pair. For example, in case of the Llama-2 model, the vocabulary consists of ~32 000 tokens.
- **embedding**: a vector representation of a single token (e.g., word2vec). These vector representations are learned in a way optimizing the next token prediction task and, intuitively, can be understood as approximating (some aspects of) the meaning of a word.
- **batch**: a set of input sampes (i.e., texts) that is passed through the LM during training simultaneously, in parallel, during one training step, before updating the internal model parameters. The **batch size** refers to the number of input samples in the set. The batch size is a common hyperparameter of the LM architectures and might have a significant effect on set up requirements (a large batch size requires a lot of memory) and the model performance (because model parameters are updated based on the training signal from the entire batch).
- **epoch**: an interation over the entire training dataset. Often a model is trained for several epochs, i.e., training iterates over the training set several times.

We will likely extend this list and learn about more important aspects as we go on with the class, but this should already equip you with a good foundation for understanding the parts of the LM literature related to data.

Main training data processing steps

Before beginning to train the LM, the following steps are commonly completed:

1. acquiring the training data: this step involves downloading or collecting the data of your choice onto the machine which will be used for training. Nowadays many datasets for various tasks can be downloaded from HuggingFace or are made available in GitHub repositories.
2. exploring and understanding the dataset: it is important to understand what kinds of texts, from which sources, on which topics, with what sentence length, ... the dataset contains. This is crucial because the model will pick up on features of the dataset in a way that might be difficult to fully anticipate (which is good if the features are, e.g., gramamticality of sentences, but bad if it is toxic language).
3. creating the desired combination: nowadays training datasets might consist of a mix of different smaller datasets. See the exercise below for more details.
4. cleaning: this step involves filtering out or converting non-machine readable or undesired characters, often lower-casing, removal of punctuation or digits or so-called stop-words (very common words like "a", "and"). However, the last three steps are not very common any more for state-of-the-art LLM training.
5. splitting the dataset into train, validation, test splits

6. prepairing the training split: training texts are often shuffled and sometimes split into shorter texts. Specifically, splitting is required if the length of a text exceeds the maximal *context window size* of the transformer model (i.e., the maximal number of tokens a model can process). In this case, texts are often split into shorter slightly overlapping chunks.
7. tokenizing: converting the single texts into lists of tokens, i.e., into lists of numerical IDs. More on tokenization in the session of week 3.
8. batching: to speed up training, the model is often fed multiple texts at the same time (i.e., at each training step). To create these batches, often additional steps are needed to ensure that several tokenized texts (i.e., several lists with token IDs) can be represented as one input tensor. These steps are either restricting texts to a maximal common length (and cutting off the rest) or *padding* all the texts to the same length. More on this in the tokenization session.

This article provides a great more detailed overview of the steps 1-4 and provides insights into traditional approaches (e.g., feature engineering) which are more common for task-specific models than for foundation language models.

> Exercise 1.1.: Massaging a Twitter dataset
>
> Below are a few code blocks for implementing some data processing steps on an example dataset of tweets about financial news, downloaded from HuggingFace. We will use the `datasets` package to work with the dataset. Originally, the dataset is intended for sentiment classification, but we will just use the tweets from the column "text".

1. Please go through the code and complete it in slots which say "#### YOUR CODE HERE". Refer to the comments and hints for instructions about what the code is supposed to do. Make sure to try to understand every line!
2. What is prominent about the dataset? Are the processing steps adequate if you wanted to train a Twitter bot which could write tweets on this data?

```
from datasets import load_dataset
import matplotlib.pyplot as plt

# 1. load dataset, only training split
dataset = load_dataset(
    "zeroshot/twitter-financial-news-sentiment",
    split="train",
)

# 2. understand dataset
# print first 5 examples
print(dataset[:5])

# print the columns of the dataset
print(dataset.column_names)

# get the number of examples in the dataset
dataset_size = ### YOUR CODE HERE ###
print(f"Dataset size: {dataset_size}")
```

```python
# compute the tweet lengths (in words, i.e., split by whitespace)
# plot them and compute the average tweet length
tweets = dataset["text"]
tweet_lengths = ### YOUR CODE HERE ###
average_tweet_length = ### YOUR CODE HERE ###
print(f"Average tweet length: {average_tweet_length}")

# plot a histogram of the tweet lengths
### YOUR CODE HERE ###
plt.xlabel("Tweet length")

# 4. clean tweets: remove non-alphabetic characters
# Hint: you can easily google how to remove non-alphabetic characters
in Python


def clean_tweet(tweet):
    """
    Remove non-alphabetic or non-space characters from a tweet.

    Args
    ----
    tweet: str
        Tweet to clean.

    Returns
    -------
    cleaned_tweet: str
        Cleaned tweet without non-alphabetic symbols.
    """
    tweet = "".join(
        ### YOUR CODE HERE ###
    )
    return tweet

# apply the preprocessing function to all tweets
cleaned_dataset = dataset.map(
    lambda example: {
        "text": clean_tweet(example["text"])
    }
)

# look at a few examples of clean tweets
print(cleaned_dataset[:2])

# 5. split dataset into training and testing set

# select the proportion of the dataset that should be used for
training
# and the proportion that should be used for testing
```

```python
# commonly train : test is around 80:20
train_size = int(0.8 * dataset_size)  ### YOUR CODE HERE ###
test_size = ### YOUR CODE HERE ###

print(f"Train size: {train_size}, Test size: {test_size}")

# split the dataset into training and testing set
# this will create two new sub-datasets with the keys "train" and
"test"
cleaned_dataset_split = cleaned_dataset.train_test_split(
    test_size=test_size,
)

print("Train split examples: ", cleaned_dataset_split["train"][:3])
print("Test split examples: ", cleaned_dataset_split["test"][:3])

# 7-8. Tokenize and batch the dataset with wrappers provided by the
datasets package
# for tokeinization, we use the GPT-2 tokenizer (more details for what
is going on
# under the hood of these wrappers is to come in the next sessions)

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")

def tokenization(example):
    """
    Wrapper around the tokenizer to tokenize the text of an example.

    Args
    ----
    example: dict
        Example tweet from the dataset. Key "text" contains the tweet.

    Returns
    -------
    dict
        Tokenized tweet with token IDs and an attention mask.
    """
    return tokenizer(### YOUR CODE HERE ###
        )

# apply the tokenization function to the train dataset
preprocessed_train_dataset =
cleaned_dataset_split["train"].map(tokenization, batched=True)

# datasets provides a handy method to format the dataset for training
models with PyTorch
# specifically, it makes sure that dataset samples that are loaded
```

```
from the
# dataset are PyTorch tensors. It also selects columns to be used.
preprocessed_train_dataset.set_format(
    type="torch",
    columns=["input_ids", "attention_mask", "label"]
)

preprocessed_train_dataset.format['type']

# finally, to see what the preprocessed dataset looks like
# we iterate over the dataset for a few steps, as we would do in
training
# note: usually a DataLoader would be used to iterate over the dataset
for training
# we will cover this in the next sessions

for i in range(5):
    print(preprocessed_train_dataset[i])
```

NOTE: if you are building your own dataset instead of e.g. loading it via `datasets`, PyTorch provides a class `Dataset` which is easily customizable and essentially allows to explicitly implement functionality that is tucked away in the `datasets` package. Working with it is covered in sheet 2.3 (for next week!).

> Exercise 1.2.: Understanding The Pile
>
> To make things more specific, consider The Pile dataset (Gao et al., 2020). Read trough the abstract and section 1 (Introduction), look at Table 1 (if needed, glimpse at other sections describing what the single names stand for), read section 5.2. The following exercises are meant to foster understanding and critical thinking about training datasets. Please try to answer the following questions to yourself:

1. Which language(s) does The Pile mainly consist of? If an LM is trained on The Pile as it is, how would you expect the LM will perform when completing a text in, e.g., Hungarian?
2. What is the difference between The Pile and Common Crawl? Why was The Pile introduced?
3. What does the "epochs" column in Table 1 refer to? What is the idea behind it?
4. What kind of data is missing from the mix reported in Table 1? Would do you think the effect of adding such data would be on an LM trained with the data?

Dataset documentation

Although datasets are a crucial part of the NLP pipeline, unfortunately, there are very few or no established practices for *documenting* shared datasets or *reporting* the datasets which are used to traing published models. This results in issues of reproducibility of the training because details about the data are unknown, biases of models due to under- or misrepresentation in the data and other issues. This paper (a completely optional read) provides an overview as well as suggestions for improving the situation in the area of machine learning:

Jo & Gebru (2020). Lessons from Archives: Strategies for Collecting Sociocultural Data in Machine Learning