Sheet 2.5: Introduction to HuggingFace & LMs

Contents

Author: Polina Tsvilodub

Introduction: ML models

In the previous sheets, we have learned how to explicitly define neural networks and recurrent language models using native PyTorch. Further, we have seen the core steps of machine learning workflows with any kind of model, in general:

1. prepare data (sheet 1.1, respective steps in sheets 2.2-2.4)
2. define model architecture (i.e., define which layers with which parameters the network should have)
3. train model on data
4. evaluate model (sheet 2.4; more on evaluation of language models to come in future sessions)

The contents of the single steps differ by application (e.g., we use different kinds of data nad models to train an image classifier vs. to train a language model), but the general structure of the process is the same.

With this background, in principle, you should be able to put together any well-known neural network yourself! (e.g., implement Llama-3 or GPT-3)! (assuming sufficient time and access to information like PyTorch docs, details of the architecture, maybe some concepts from the next lecture, of course)

However, putting together the network (i.e., specifying all the layers, their sizes, implementing the forward pass through the net etc.) does *not* mean that you will be able to use the model for generating predictions in practice. Generating predictions (= running inference) on new inputs (e.g., predicting continuations of new input texts or classifying new images) is the purpose of creating a machine learning model – we build them because we want to automate these tasks. Ideally, we want the model to *generalize well* – to be able to handle inputs for that the model was not trained on and still perform the task correctly (e.g., produce a sensible text completion for an entirely new text input).

For generating predictions, we need the **trained** model, i.e., we need not just the definition of the model, but also the *model weights* set to optimized values that result from (successful) training. That is, we need the values of all the weight matrices (we have seen those in sheet 2.3). In practice, successful training of state-of-the-art models is quite costly and data-hungry (at least for creating actually useful models).

NB: The practical sheets will try to be consistent and use the term “architecture” to refer to the more abstract definition, carcass, of a model (i.e,. the specific layer types and parameterizations underlying the model), and the term “model” for the particular instatiation with particular weights of that model configuration. When referring to a specific model instance, there are different important concepts / terms:

- **pretrained** model: this is commonly used in the domains of NLP and computer vision (CV) where models are usually first trained on a very large collection of data to learn their task (in general). In the context of NLP, this refers to a model that has been trained to predict the next word on a large collection of text (e.g., on the Pile which we have seen in sheet 1.1). On a high level, this step can be thought of as the step where the model learns to predict fluent language in general. In the context of CV, this often refers to a model that has been trained to classify images on the ImageNet dataset.
- **fine-tuned** model: this refers to a pretrained model that has been further trained on a *task-specific*, or, generally curated and therefore, usually smaller, dataset. This step can be thought of as optimizing the model for a particular task, like predicting texts in a particular domain (e.g., movie reviews). This is usually done based on a pretrained model

Skip to main content

The term “trained” model will be mostly used as an umbrella terms for both types of models.

> **Exercise 2.5.1: ML models workflow**
>
> 1. Given examples from the previous sheets, in your own words, describe the difference between *training* a model and running *inference* with a model. (Understanding this is absolutely critical; please ask questions if have any doubts!)
> 2. Suppose you want to train the model GPT-2 to generate IMDB movie reviews. In your own words (few intuitive sentences), describe the highlevel steps that you would do to accomplish this task.

HuggingFace 🤗

Luckily, there are many popular *open-source* models, i.e., those for which the architecture and the weights have been made freely accessible by the developers: e.g., GPT-2, the LLama models, BERT etc. In contrast, e.g., GPT-4 is *closed-source* – we don’t know exactly which layers with which configurations are inside the model and we cannot access the weights ourselves; they are only hosted by OpenAI and we can send our input data to their servers and get back predictions (by using the OpenAI API).

Open-source models are frequently used by the community and have been made available via platforms and packages which provide an easy interface for loading pretrained models, generating predictions from these pretrained models, or training models from scratch. The by-far most used platform for open-source resources which we will heavily use in this course is **HuggingFace**. HuggingFace provides infrastructure for working with many different types of machine learning models (for working with audio, vision, language tasks), but, as you might guess, we will be focusing on functionality relevant to language modeling. HuggingFace will often be abbreviated as HF throughout the sheets.

Here is a high-level overview of all the nice things that HuggingFace provides:

- (most relevant for us) the packages **`transformers`** and **`datasets`** which we already installed alst week. These allow easy access and workflow with models and with datasets.
  - Overviews, guides, tutorials for different tasks that can be done with `transformers`, docs for different models as well as the documentation of the API can be found here.
  - The same resources for `datasets` can be found here.
- The (online) HF platform called HuggingFace Hub actually stores trained model weights (also sometimes called checkpoints) and datasets. They are downloaded to the local machine when accessed through the `transformers` or `datasets` package. Available models can be browsed here; available datasets can be browsed here.
  - In order to *upload* yourown resources to HF Hub and in order to access some models (inclding LLama-2), you need a free account. It can easily be created on HF. So far, an account is not required for following the class.
- The platform provides much more (feel free to explore! but also try not to feel overwhlemed if you are seeing HF for the first time). Here are some highlights most relevant to the course:
  - NLP course which includes practical tutorials for common NLP tasks with transformers.
  - **`evaluate`** package for computing common evaluation metrics (more on this in the next sessions)
  - (more advanced) packages for accelerated and parallelized training on GPUs, via parameter-efficient fine-tuning (PEFT) and more.
  - `tokenizers` package. However, we will only use functionality of pretrained tokenizers which are shipped together with models via transformers.

> **Exercise 2.5.2: Familiarization with HuggingFace**
>
> 1. Just for yourself, try to find whether the model BERT is available on HF. If yes, try get a sense of the kind of information that HF provides about available models.

Skip to main content

Working with LMs via 🤗 Transformers

We will dive right into using the `transformers` package ourselves. The learning goals for the remaining part of the sheett are:

- learn how to load a pretrained language model
- learn how to generate a prediction with the model
- learn about the `Trainer` class which allows to train from scratch or fine-tune the model (the following code is inspired by this tutorial)

NB: this sheet uses the pretrained GPT-2, a causal (decoder-only) transformer model. We have not covered transformers in detail yet, so the set up of the model and some of the required set-up for using it are treated as a blackbox in this sheet. More in-depth explanations will follow the next lecture covering transformers.

There are two ways of generating predictions with a trained model available on HF. First, we will do so explicitly. The steps are:

- load the pretrained model with its dedicated class
- tokenize some input text
- generate predictions with greedy decoding (i.e., under the hood, compute forward passes through the model and retrieve the most likely token, given the previous input)
- convert the newly generated tokens back into human-readable strings

```
# import packages
# note: if you are on Colab, you might need to install some requirements
# as we did in Sheet 1.1. Otherwise, don't forget to activate your local environment

import transformers
from transformers import pipeline
from transformers import GPT2Tokenizer, GPT2LMHeadModel, AutoTokenizer, AutoModelForCausalLM
from datasets import load_dataset
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
import torch
import matplotlib.pyplot as plt
from transformers import DataCollatorForLanguageModeling
from transformers import Trainer, TrainingArguments
```

```
# additioanlly, we need to install accelerate
# uncomment and run the following line on Colab or in your environment
# !pip install accelerate
# NOTE: in a notebook, reloading of the kernel might be required after installation if you get dependenc
```

```
# define computational device
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}")
```

```
# define input text
input_text = "Natural language processing is "

# load pretrained model and respective tokenizer
# the class GPT2LMHeadModel initialized the "container" for the GPT-2
# architecture with an LM head - i.e., an output layer for predicting next tokens
# .from_pretrained loads the pretrained weights into the container
model = GPT2LMHeadModel.from_pretrained("gpt2").to(device)
# load pretrained tokenizer belonging to GPT-2 (more on tokenizers next session)
##### YOUR CODE HERE ##### (Hint: you have seen how to do this in sheet 1.1)
tokenizer =

# tokenize our input text
input_ids = tokenizer.encode(input_text, return_tensors="pt").to(device)
print("Tokenized text (input IDs) ", input_ids)
```

Skip to main content

```
# decode the prediction (i.e., convert tokens back to text)
answer = tokenizer.decode(prediction[0])
print("Predicted continuation: ", answer)
```

Alternatively, we can make use of transformer's `pipeline` – a utility wrapper around these steps which makes it easier for us to generate predictions, without having to manually implement them. HF provides pipelines for many common tasks.

```
# initialize the pipeline with the model of our choice
generator = pipeline(
    # specify task
    'text-generation',
    model='gpt2'
)

# run the pipeline with default configs
greedy_pipe_output = generator(input_text)
print("Greedy pipeline output: ", greedy_pipe_output)
```

Exercise 2.5.3: Generating predictions

1. Come up with a different input string and generate a prediction. Would you consider the output a good prediction?
2. Change the code (explicit or implicit) by adding a few parameters so as to use *pure sampling* for predicting the output. How do the predictions change? (Hint: see slides for a reminder about decoding schemes and docs for practical implementation)

Another common use of pretrained models is to further fine-tune them on a specific dataset, e.g., to specialize them on generating text of a particular domain, or to train them for a specific task. The latter is often with a task-specific output *head* – i.e., a different output layer is used. For instance, for classification tasks (like sentiment classification), a classification layer could be trained on a sentiment classification dataset, while keeping the rest of the network fixed from pretraining (or, frozen).

The following part shows how to fine-tune GPT-2 on a specific corpus – IMDB movie reviews. Specifically, we will make use of the transformers `Trainer` class which provides convenient wrappers around different steps that need to be done in order to train a model. It essentially simplifies the implementation of the single substeps that are summarized under step 3 in the high-level overview of the ML workflow at the beginning of the sheet.

Exercise 2.5.4: Training steps

1. As a reminder, using pseudo-code, list of concrete steps that constitute a single training step in a training loop. Take a look at the training loops in sheets 2.2 and 2.3 to compare your results.
2. As you go through the following code, please complete it in spots which are commented with "### YOUR CODE HERE ###"

```
# load IMDB dataset
# dataset = load_dataset("truthful_qa", "generation")
dataset = load_dataset("stanfordnlp/imdb", split="train")
```

```
# set the pad token of the tokenizer which we loaded above
tokenizer.pad_token = tokenizer.eos_token
# set the side on which pad tokens should be added to the input (if required)
tokenizer.padding_side = "left"
# define a function to collate the data into batches and tokenize it
# this helper allows to massage the data such that it can be batched into the same
# tensor representing a batch of inputs and can be used for training.
# specifically it creates *labels* for the model to predict
data_collator = DataCollatorForLanguageModeling(tokenizer, mlm=False)

def tokenize(
        input_text,
        max_length=64,
        tokenizer=tokenizer
    ):
    """
   YOUR TEXT HERE.
```

Skip to main content

```
    ----------
    input_text : YOUR TEXT HERE.
    max_length : YOUR TEXT HERE.

    Returns
    -------
    YOUR TEXT HERE.
    """
    tokenized_input = tokenizer(
        input_text["text"],
        max_length=max_length,
        padding="max_length",
        truncation=True,
        return_tensors="pt"
    )
    # mask padding tokens so that the model doesn't attend to them
    # more on masking next week
    tokenized_input["attention_mask"] = (tokenized_input["input_ids"] != tokenizer.pad_token_id).long()
    return {"input_ids": tokenized_input['input_ids'],
            "attention_mask": tokenized_input['attention_mask']}

tokenized_datasets = dataset.map(
    tokenize, batched=True, remove_columns=dataset.column_names
)
```

```
# print a sample of the tokenized dataset
tokenized_datasets[0]
```

```
# since we don't want to fully fine-tune the model but rather learn
# the basics of the set up, we will only use a small subset of the data
subsampled_dataset = tokenized_datasets.select(range(500))
```

```
# we inspect what the data_collator does
out = data_collator([subsampled_dataset[i] for i in range(5)])
for key in out:
    print(f"Tensor {key}: {out[key][0]}")
```

IMPORTANT: it is key to understand the language modeling objective (i.e., mechanism used to train a model to predict the next token) and how it is implemented in practice. Conceptually, the LM objective is discussed at the beginning of the tutorial. In terms of implementation, the addition of special tokens and shifting happens within each transformers model. You can find and example fro GPT-2 here. As you can see, the collator function above creates the labels that are used as target for prediction for us. Please make sure that you understand this step and that you would be able to create labels, given an input, yourself.

Exercise 2.5.5: Understanding Trainer

The next cell defines the key configurations of the training in `TrainingArguments`. These are passed to the `Trainer` in order to instantiate a training loop.

1. Please use the documentation here to write short comments for each relevant line of the following code cell, so that you know what different training parameters there are.
2. What is the size of the training data? What is the size of the test data?

```
# define training arguments
args = TrainingArguments(
    output_dir="imdb_gpt2",
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    evaluation_strategy="steps",
    eval_steps=5,
    logging_steps=1,
    gradient_accumulation_steps=1,
    num_train_epochs=1,
    weight_decay=0.1,
    lr_scheduler_type="cosine",
    learning_rate=5e-4,
    save_steps=5_000,
    fp16=True if device == "cuda" else False,
    push_to_hub=False,
    use_mps_device=True if device == "mps" else False
```

Skip to main content

```
trainer = Trainer(
    model=model,
    tokenizer=tokenizer,
    args=args,
    data_collator=data_collator,
    train_dataset=subsampled_dataset,
    eval_dataset=subsampled_dataset,
)
```

```
# run the training of the model
trainer.train()
```

Note that we did not explicitly pass a loss function to the training. The Trainer constructs it automatically. However, we could also explicitly define the language modeling loss. For this, we need to override the `compute_loss()` method of the Trainer and build a custom Trainer with it.

```
class IMDBTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False):
        outputs = model(**inputs)
        loss = outputs.loss
        return (loss, outputs) if return_outputs else loss
```

Exercise 2.5.6: CrossEntropyLoss [more advanced]

1. Implement an even more explicit definition of the loss function by using the `torch.nn.CrossEntropyLoss()` and the labels which are passed as part of the inputs instead of using outputs.loss. The respective docs can be found on the PyTorch website. NB: Make sure to correctly shift your labels.

```
# use explicit trainer
explicit_trainer = IMDBTrainer(
    model=model,
    tokenizer=tokenizer,
    args=args,
    data_collator=data_collator,
    train_dataset=subsampled_dataset,
    eval_dataset=subsampled_dataset,
)

explicit_trainer.train()
```

Finally, we want to inspect the training dynamics by plotting the training and evaluation losses. These can be accessed in the Trainer history. Ideally, both of them should decrease – this would indicate successful optimization. Feel free to play around with training parameters and plot the losses to see what effects different values have.

```
# plot the train and evaluation losses
log_history = trainer.state.log_history

# wrangle the trainer logs into a dataframe for easier plotting
train_losses = []
test_losses = []
for d in log_history:
    if "loss" in d.keys():
        train_losses.append(d["loss"])
    elif "eval_loss" in d.keys():
        test_losses.append(d["eval_loss"])
```

```
# plot
plt.plot(train_losses, label="train loss")
plt.plot(test_losses, label="test loss")
plt.xlabel("Logged steps")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

Skip to main content

Outlook

The code above used a GPT-2 specific class, `GPT2LMHeadModel`, to load the model. However, instead of the architecture-specific classes the so called AutoClass can often be used instead. It essentially “infers” the architecture from a configuration file that is retrieved when a model is downloaded from HF. In the next tutorials, you will often see subclasses of this being used. You can find more (optional) information here.

Previous
‹ Sheet 2.4: Character-level sequence modeling w/ RNNs