02_get_familiar_with_train

April 2, 2024

[ ]:
```
import os, sys
from pprint import pprint
from transformers import AutoTokenizer

%load_ext autoreload
%autoreload 2

sys.path.append("../cli")  # make all python files from the directory cli/␣
 ↪importable
import train as train_script
```

```
c:\Users\Nech\anaconda3\envs\comp5500-hw6\Lib\site-packages\tqdm\auto.py:21:
TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
c:\Users\Nech\projects\python_projects\comp5500\hw6\hw6_transformer_mt\notebooks
\../cli\train.py:62: FutureWarning: load_metric is deprecated and will be
removed in the next major version of datasets. Use 'evaluate.load' instead, from
the new library  Evaluate: https://huggingface.co/docs/evaluate
  bleu = datasets.load_metric("sacrebleu")
c:\Users\Nech\anaconda3\envs\comp5500-hw6\Lib\site-
packages\datasets\load.py:756: FutureWarning: The repository for sacrebleu
contains custom code which must be executed to correctly load the metric. You
can inspect the repository content at https://raw.githubusercontent.com/huggingf
ace/datasets/2.18.0/metrics/sacrebleu/sacrebleu.py
You can avoid this message in future by passing the argument
`trust_remote_code=True`.
Passing `trust_remote_code=True` will be mandatory to load this metric from the
next major release of `datasets`.
  warnings.warn(
```

1 Training script

Every training script roughly follows the same idea:

1. Load the data
2. Create the model and load the tokenizers
3. Pre-process the data

4. Create PyTorch dataloaders that handle data shuffling and batching
5. Create optimizer and (optionally) learning rate scheduler
6. Training loop
7. Evaluation loop
8. Save the model

Some of these steps are usually simple (e.g., saving the model), some are usually complicated (e.g., pre-processing the data), but many of them really depends on what you are trying to achieve. For example, in most cases, the training loop is very standard, but as soon as you want to control your training more it becomes more and more complicated. The usual things you may want to add to training are early stopping, multi-GPU support, or just more metrics.

Nevertheless, you will always see all of these steps in every deep learning project, so let's look at them closer.

1.1 1. Load the data

In our case, it is a very simple task. We use Datasets library, which downloads the data files from the hub and provides us with a `Dataset` object. You can also use the same `load_dataset` to load your local files (not datasets form the hub) in `.csv` or `.json` formats.

This is what this part looks like in our training script. Note that we check if the dataset has a validation split, and if it doesn't we create one.

```
raw_datasets = load_dataset(args.dataset_name, args.dataset_config_name)
if "validation" not in raw_datasets:
    # will create "train" and "test" subsets
    # fix seed to make sure that the split is reproducible
    # note that we should use the same seed here and in create_tokenizer.py
    raw_datasets = raw_datasets["train"].train_test_split(test_size=2000, seed=42)
```

Additionally, when developing the model, it can be very useful to work not with the whole dataset, but with a very small sample of it (maybe just 100 examples). This way your training loop will be quick and you can quickly iterate on your code and fix all the bugs.

```
if args.debug:
    raw_datasets = utils.sample_small_debug_dataset(raw_datasets)
```

You can find the function `sample_small_debug_dataset()` in `transformer_mt/utils.py`.

[ ]:
```
done = True
assert done, "please read the instructions above and then change the value of␣
 ↪done to True"
```

1.2 2. Create the model and load the tokenizers

Here we create `PreTrainedTokenizerFast` and `TransfomerEncoderDecoderModel` objects that we will use throughout the rest of the script. Because this is a machine translation task, we have two tokenizers: one for the language we are translating **from** and the other for the language we are translating **to**.

```
source_tokenizer = ...
source_tokenizer = ...

model = TransfomerEncoderDecoderModel(...)
```

Now, go to `train.py` and complete tasks 4.1 and 4.2.

[ ]:
```
done = True
assert done, "please complete the task described above and then change the value of done to True"
```

1.3 3. Pre-process the data

This is usually a very annoying to code part of the script. This is why we did it for you!

For the translation task, we need to 1. Tokenize source and target texts with the corresponding tokenizer. 1. Truncate them to the maximum length we want our model to work with. 1. Shift decoder inputs to the left from the target values, so that decoder would learn to predict the next word of the translation given previous words.

> Now, answer inline questions 4.1 and 4.2 in `preprocess_function()`. Please write your answers in `train.py` and **not** in this notebook.

Try out `preprocess_function` and feel free to play with its input. Notice that `decoder_input_ids` are almost exactly like `labels`, but have a special beginning-of-sentence token at the first position. During training, we always know the translation in advance and we train our system just like a language model. But when the model is trained and we translate a sentence without knowing its translation in advance, we always input the beginning-of-sentence (BOS) token into the decoder to produce the first word of the translation. Then we input `BOS first_word` to produce a second word, we input `BOS first_word second_word` to produce the third word, and so on.

[ ]:
```
examples = {"translation": [
    {"en": "Hello", "fr": "Bonjour"},
    {"en": "How are you?", "fr": "Comment allez-vous?"},
]}
source_lang = "en"
target_lang = "fr"
max_seq_length = 128

# ignore that we use the same tokeinzer for both languages, this is just for demonstration
source_tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
target_tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
target_tokenizer.add_special_tokens({"bos_token": "<bos>"})
target_tokenizer.add_special_tokens({"eos_token": "<eos>"})

preprocessed = train_script.preprocess_function(
    examples=examples,
    source_lang=source_lang,
    target_lang=target_lang,
```

```
    source_tokenizer=source_tokenizer,
    target_tokenizer=target_tokenizer,
    max_seq_length=max_seq_length
)
pprint(preprocessed)

done = True
assert done, "please complete the task described above and then change the␣
 ↪value of done to True"
```

```
{'attention_mask': [[1, 1, 1], [1, 1, 1, 1, 1, 1]],
 'decoder_input_ids': [[119547, 101, 30120, 98214, 10129, 102],
                       [119547,
                        101,
                        105415,
                        10968,
                        10305,
                        118,
                        24931,
                        136,
                        102]],
 'input_ids': [[101, 31178, 102], [101, 14962, 10301, 13028, 136, 102]],
 'labels': [[101, 30120, 98214, 10129, 102, 119548],
            [101, 105415, 10968, 10305, 118, 24931, 136, 102, 119548]],
 'token_type_ids': [[0, 0, 0], [0, 0, 0, 0, 0, 0]]}
```

1.4 4. Create PyTorch dataloaders that handle data shuffling and batching

When the data is pre-processed we need to be able to collate it into batches and we need to do it quickly. The problem is that any non-GPU computation you do during training slows you down. We don't want our GPUs to wait until the data is ready and this is why PyTorch provides us with DataLoaders, which can use multiple CPU cores to quickly read the data, (optionally) pre-process it, pad it to a fixed length, and combine into a batch.

You can read more about dataloaders in the official PyTorch documentation.

Padding functions can be annoying to write too because there are so many ways to do them. For this homework we wrote a relatively inefficient, but easy-to-understand padding function `utils.pad()` and a function `collation_function_for_seq2seq()` that accepts a list of pre-processed examples, pads them to the maximum length of these examples and creates batches.

```
def collation_function_for_seq2seq(batch, source_pad_token_id, target_pad_token_id):
    """
    Args:
        batch: a list of dicts of numpy arrays with keys
            input_ids
            decoder_input_ids
            labels
    """
    input_ids_list = [ex["input_ids"] for ex in batch]
```

```
    decoder_input_ids_list = [ex["decoder_input_ids"] for ex in batch]
    labels_list = [ex["labels"] for ex in batch]

    collated_batch = {
        "input_ids": utils.pad(input_ids_list, source_pad_token_id),
        "decoder_input_ids": utils.pad(decoder_input_ids_list, target_pad_token_id),
        "labels": utils.pad(labels_list, target_pad_token_id),
    }

    collated_batch["encoder_padding_mask"] = collated_batch["input_ids"] == source_pad_token_id
    return collated_batch
```

Your next step is to code task 4.3

[ ]:
```
batch = [
    {"input_ids": [101, 31178, 102], "decoder_input_ids": [119547, 101, 30120, 98214, 10129, 102], "labels": [101, 30120, 98214, 10129, 102, 119548]},
    {"input_ids": [101, 14962, 10301, 13028, 136, 102], "decoder_input_ids": [119547, 101, 105415, 10968, 10305, 118, 24931, 136, 102], "labels": [101, 105415, 10968, 10305, 118, 24931, 136, 102, 119548]},
]
collated_batch = train_script.collation_function_for_seq2seq(
    batch,
    source_pad_token_id=0,
    target_pad_token_id=1,
)

pprint(collated_batch)

done = True
assert done, "please complete the task described above and then change the value of done to True"
```

```
{'decoder_input_ids': tensor([[119547,    101,  30120,  98214,  10129,    102,
1,      1,      1],
        [119547,    101, 105415,  10968,  10305,    118,  24931,    136,
102]]),
 'encoder_padding_mask': tensor([[False, False, False,  True,  True,  True],
        [False, False, False, False, False, False]]),
 'input_ids': tensor([[  101, 31178,   102,     0,     0,     0],
        [  101, 14962, 10301, 13028,   136,   102]]),
 'labels': tensor([[   101,  30120,  98214,  10129,    102, 119548,      1,
1,      1],
        [   101, 105415,  10968,  10305,    118,  24931,    136,    102,
119548]])}
```

1.5 5. Create optimizer and (optionally) learning rate scheduler

Usually a straightforward step. If you are using something simple like ADAM, just pass model parameters, learning rate, and any other extra arguments.

```
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=args.learning_rate,
    weight_decay=args.weight_decay,
)
```

A scheduler is an optional thing, but they work really well with Transformers. You can read more about schedulers here https://huggingface.co/docs/transformers/main_classes/optimizer_schedules transformers.get_scheduler is a convenience function that accepts the scheduler name and returns a function that changes the optimizer learning rate according to the schedule every time we call lr_scheduler.step()

```
lr_scheduler = transformers.get_scheduler(
    name=args.lr_scheduler_type,
    optimizer=optimizer,
    num_warmup_steps=args.num_warmup_steps,
    num_training_steps=args.max_train_steps,
)
```

[ ]:
```
done = True
assert done, "please read the instructions above and then change the value of␣
↪done to True"
```

1.6 6. Training loop

Training loops can be arbitrary complex, but if we stick to the simplest one for our task, it would look roughly like this.

```
for epoch in range(args.num_train_epochs):
    for batch in train_dataloader:
        input_ids = batch["input_ids"].to(args.device)
        decoder_input_ids = batch["decoder_input_ids"].to(args.device)
        key_padding_mask = batch["encoder_padding_mask"].to(args.device)
        labels = batch["labels"].to(args.device)

        logits = model(
            input_ids,
            decoder_input_ids=decoder_input_ids,
            key_padding_mask=key_padding_mask,
        )

        loss = F.cross_entropy(
            logits.view(-1, logits.shape[-1]),
            labels.view(-1),
```

```
            ignore_index=target_tokenizer.pad_token_id,
        )

        loss.backward()
        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()

        wandb.log({
            "train_loss": loss,
            "learning_rate": optimizer.param_groups[0]["lr"],
            "epoch": epoch,
            },
            step=global_step,
        )
```

Here, every batch is an object returned by a `collate_fn` function.

1. Move all of the tensors you will use for your neural network input and loss calculation to the GPU.
2. Produce logits with your model Remember that sequence-to-sequence model accepts

- input_ids (encoder input, the sequence we want to translate)
- decoder_input_ids (decoder input, the translation shifted to the left)
- key_padding_mask (for masking out PAD tokens in encoder input)

3. Use F.cross_entropy to compute the loss. Notice that you might need to reshape the tensors to do that into [batch_size * sequence_length, vocab_size] and reshape labels into [batch_size * sequence_length] Ignore target_tokenizer.pad_token_id in loss computation (argument ignore_index).
4. Compute the loss gradients with .backward()
5. Update the parameters
6. Update the learning rate using the scheduler
7. Zero out the gradients so that they don't accumulate between steps

[ ]:
```
done = True
assert done, "please read the instructions above and then change the value of␣
↪done to True"
```

1.7 7. Evaluation loop

Usually, evaluation is either performed at the end of an epoch or every n training iterations. The second approach is preferred if you have a very large training set for which one epoch can take hours (as we have in this homework).

This is what it schematically looks like:

```
global_step = 0
for epoch in range(args.num_train_epochs):
    for batch in train_dataloader:
        # training loop stuff
```

```
...

global_step += 1
if global_step % args.eval_every_steps == 0 or global_step == args.max_train_steps
    eval_results, decoded_preds, decoded_labels = evaluate_model(
        model=model,
        eval_dataloader=eval_dataloader,
        ...
    )
    wandb.log(eval_results)
```

Evaluation of sequence-to-sequence models is quite different from the usual evaluation loop. Compared to the classification task, where you just need to produce a single number (class index) given the input, for generation tasks, you need to produce a sequence of numbers (token indices).

And if during training we know them all in advance and insert `decoder_input_ids`, this is not how we want (or can) use these models in real life. We talked about greedy generation and beam search in the class, you can find their implementations in `transformer_mt/modeling_transformer.py`. Specially `TransfomerEncoderDecoderModel._generate_greedy` and `TransfomerEncoderDecoderModel._generate_beam_search`.

As you may see, these functions are quite different in complexity, but the results of writing the beam search are rewarding. Sometimes the difference can be as large as 5 BLEU points. Feel free to compare your model performance with `beam_size=5` and `beam_size=1` (greedy generation).

> Look into `evaluate_model()` in `train.py` and answer an inline question.

1.8 8. Save the model

To save the model, we need to save two entities: model weights which are also called, a checkpoint or a `state_dict`, and model config. We need this because to restore the model we need to first create a model object (with any parameters) and then use `.load_state_dict()` to replace the existing weights with the ones from the checkpoint.

```
logger.info("Saving final model checkpoint to %s", args.output_dir)
model.save_pretrained(args.output_dir)

logger.info("Uploading tokenizer, model and config to wandb")
wandb.save(os.path.join(args.output_dir, "*"))
```

Additionally, we upload everything to WandB, so that you could download a trained model from the website later if you need it and didn't have to store it on your machine or in the cloud.

To see how `.save_pretrained()` and `.from_pretrained()` are implemented in this homework (which is significantly more simple compared to Transformers implementation of similar functions), feel free to look at `transformer_mt/modeling_transformer.py`.