A comparing guide to train language models with Python.

Lennart Keller

Nov 29, 2021

CONTENTS

For the sake of readability, the listings in this text are shortened if necessary. To take a look at the whole code, please visit the associated GitHub-Repository. Note that the dataset has to be downloaded manually after registration at https://cs.rochester.edu/nlp/rocstories/.

Just place the CSV-File `ROCStories_winter2017 - ROCStories_winter2017.csv` in the `scripts/data` directory and run the `BuildRocStoriesDataset.ipynb` notebook to create the dataset.

To run the actual experiments, execute the bash scripts from within their directories.

To run all experiments at once, just run the `run_all.sh` script from within its location

Execution Summary

Document	Modified	Method	Run Time (s)	Status
HuggingFaceTrainer	2021-11-29 16:41	cache	4.03	✅
HuggingfaceEcosystem	2021-11-29 16:41	cache	12.22	✅
Poutyne	2021-11-29 16:41	cache	12.08	✅
Prerequisites	2021-11-29 16:41	cache	17.1	✅
PyTorchLightning	2021-11-29 16:41	cache	1.29	✅

CHAPTER

ONE

INTRODUCTION

Transformer-based neural language models have revolutionized the world of NLP. Their ability to process long-range dependencies within texts and gain language processing abilities via self-supervised pretraining allowed them to set-state-of the art results across many different tasks. These successes made them a technique with great interest across many disciplines of academia and the industry alike.

But training a transformer can be challenging. One reason for this is the state of the software landscape. The high rate of innovations in this field drives the development of new models and architectures. Software libraries trying to keep up with this rapid pace have to regularly adapt to new models and techniques, complicating building robust and stable software. The other reason lies in the complexity of the models. Like any other neural network, training a transformer bases mode is a complicated process that requires a lot of technical and domain knowledge. Also, a robust understanding of how neural networks work and what pitfalls must be avoided is needed. While no software can replace domain knowledge and general understanding, numerous frameworks aim at lowering the barrier of entry by mitigating the technical complexities. Depending on the scale of the network, these technical hurdles may include things like multi-GPU training, data loading, hyperparameter search, progress tracking, early stopping, and many more. The scope of this work is two-fold: Firstly, it compares three of these frameworks by using them to train a language model on predicting the correct order of a sequence of shuffled sentences. Furthermore, it also acts as a tutorial that guides through the basic features of each framework and shows how to adapt them to work with language models. As the underlying source of pretrained language models, the `transformers` library (Wolf *et al.* [2019]) from Huggingface, which has become the de-facto standard for these models, is used.

The following frameworks are chosen: The built-in `Trainer` of the `transformers` library itself, PyTorch Lightning, a framework, which offers a general framework to train neural networks of all kinds; and Poutyne, a library which tries to bring the ease of Tensorflow's Keras API to PyTorch. This selection is intended to segment the broader set of available frameworks into different levels of complexity. Du its origin, the `Trainer` is highly optimized for working with `transformers`. Thus it can be seen as the middle ground between the pro-level solution PyTorch Lightning and the more beginner-friendly Poutyne framework.

The rest of the works is structured as follows: Firstly, a brief introduction to the Huggingface software stack, which serves as the foundation for all experiments, is given. Next, the experimental design is presented. Since the frameworks only control the actual training of the models, many parts of the experiment will be mostly the same. These steps are laid out in the next chapter to avoid redundancies in the following chapters, where each library is presented, and the implementational details are discussed. Then, the results of all experiments are analyzed to check if the choice of a framework affects the model's performance. And, of course, to see if the sentence ordering works in general. Finally, we end with a concluding comparison of the frameworks carving out their strengths and weaknesses and discussing their different potential for specific use cases.

CHAPTER

TWO

THE HUGGINGFACE ECOSYSTEM

2.1 `tranformers`

In 2018 on the same day that Google published its research implementation of BERT, developed in Tensorflow, Thomas Wolf, a researcher at the NLP startup Huggingface, created a Github repository called "PyTorch-transformers." The initial goal of this project was to load the weights of the Bert model, published alongside the paper in Tensorflow, with PyTorch.

From here on, this repository quickly evolved into the Transformers library, which sits at the heart of the Huggingface NLP infrastructure. The goal of the transformers library is to provide the majority of transformer-based neural language models alongside all of the extra tooling required to use them.

Originating as a pure PyTorch library, Huggingface widened its scope over the last two years and integrated other deep learning frameworks such as Tensorflow or the newly created Flax library. But these additions are relatively unstable and subject to frequent significant changes so that this work will only focus on the much more stable PyTorch branch of the Transformers library.

2.2 `tokenizers`

A notable characteristic of modern language models is that nearly all ship with a custom, fitted tokenizer. These tokenizers operate on a subword level and are trained to represent texts with a fixed-sized vocabulary. Huggingface provides the `tokenizers` library that offers implementations of the most common tokenizer models. These tokenizers come in two versions, a fast one written in Rust and a slower python implementation. For the sake of efficiency, the Rust version is the best choice most of the time.

2.3 `datasets`

Lastly, to complete the NLP pipeline, Huggingface also develops a library for Dataset management, called `datasets`. This library aims to streamline the process of data preparation and provide a consistent interface to create, store, and process large datasets too large to fit into the memory in a whole.

With these three libraries, it is possible to cover the overwhelming majority of possible tasks.

2.4 Interoperability

To make all libraries as interoperable as possible, they use dictionaries or dictionary-like objects as a standard data exchange format. These dictionaries contain all argument names of the function or method that is supposedly called next as keys and the data as values.

```
from transformers import AutoTokenizer, AutoModel
from datasets import Dataset

model = AutoModel.from_pretrained("bert-base-cased", add_pooling_layer=False)
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

dataset = Dataset.from_dict({"text": ["Dictionaries? Everywhere!"]})

data = dataset[0]
print(data)

inputs = tokenizer(data["text"], return_tensors="pt")
print(inputs)
outputs = model(**inputs)
print(outputs)
print(outputs["last_hidden_state"])
```

```
{'text': 'Dictionaries? Everywhere!'}
{'input_ids': tensor([[  101, 12120,  5796,  5927,   136,  4081, 15839,   106,
 ↪102]]), 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0]]), 'attention_mask
 ↪': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1]])}
BaseModelOutputWithPoolingAndCrossAttentions(last_hidden_state=tensor([[[ 0.4575,
 ↪0.0958, -0.0544,  ..., -0.1948,  0.3481, -0.1724],
         [-0.5101, -0.1217,  0.8169,  ...,  0.5608,  0.3777, -0.0601],
         [-0.4231,  0.8276, -0.2315,  ..., -0.4748, -0.1374,  0.2291],
         ...,
         [-0.1184, -0.2610, -0.1218,  ..., -0.1848,  0.1142, -0.5246],
         [ 0.5114, -0.0423,  0.2668,  ...,  0.3748,  0.2570, -0.0132],
         [ 0.6647,  0.5592, -0.1306,  ..., -0.3199,  0.4948, -1.2197]]],
       grad_fn=<NativeLayerNormBackward>), pooler_output=None, hidden_states=None,
 ↪past_key_values=None, attentions=None, cross_attentions=None)
tensor([[[ 0.4575,  0.0958, -0.0544,  ..., -0.1948,  0.3481, -0.1724],
         [-0.5101, -0.1217,  0.8169,  ...,  0.5608,  0.3777, -0.0601],
         [-0.4231,  0.8276, -0.2315,  ..., -0.4748, -0.1374,  0.2291],
         ...,
         [-0.1184, -0.2610, -0.1218,  ..., -0.1848,  0.1142, -0.5246],
         [ 0.5114, -0.0423,  0.2668,  ...,  0.3748,  0.2570, -0.0132],
         [ 0.6647,  0.5592, -0.1306,  ..., -0.3199,  0.4948, -1.2197]]],
       grad_fn=<NativeLayerNormBackward>)
```

2.5 PyTorch-Backend

Relying on PyTorch as the underlying deep learning framework comes with one caveat: Unlike Tensorflow, which has integrated Keras as a high-level API for training neural networks, PyTorch does not provide any tools to facilitate the training process. Instead, PyTorch's research-orientated nature makes it entirely up to the users to implement the training loop. While this is no problem when researching and experimenting with new techniques, it is often time-consuming in the practitioner's case. When applying standard models to tasks like text classification, implementing the training loop is an obstacle that only increases development time. Also, it introduces a new space for making errors.

In most application-oriented scenarios, the training loop roughly looks like this:

```
...
model = create_model()
model.to(DEVICE)
train_data, val_data = load_data()
optimizer = torch.optim.SGD(lr=5e-5, params=model.parameters())
for train_step, batch in enumerate(train_data):
    model.train()
    input_data, targets = batch
    input_data = input_data.to(DEVICE)
    targets = targets.to(DEVICE)
    outputs = model(input_data)
    loss = loss_function(outputs, targets)

    # Compute gradients w.r.t the input data
    loss.backward()
    # Update the parameters of the model
    optimizer.step()
    # Clear the gradients before next step
    optimizer.zero_grad()

    train_log(train_step, loss)

    # Validate the performance of the model every 100 train steps
    if train_step % 100 == 0:
        model.eval()
        for val_step, batch in enumerate(val_data):
                input_data, targets = batch
                input_data = input_data.to(DEVICE)
                targets = targets.to(DEVICE)
            with torch.no_grad():
                outputs = model(input_data)
                val_loss = loss_function(outputs, targets).detach().cpu()
                # Compute other val metrics (i.e. accuracy)
                val_score = other_metric(outputs, targets)

                val_log(val_step, val_loss, val_loss)
...
```

But not only can it become quite tedious to write this loop (or variations of it) repeatedly, but more gravely, it sets a barrier of entry for beginners or non-experts because it adds another layer of complexity when tinkering around with deep learning.

Another implication of outsourcing this process to the users hits when the models grow in size. Modern language models may require a massive amount of memory even when trained with tiny batch sizes. There are strategies to overcome these limitations, like gradient accumulation. But all these tricks again have to be implemented by the user. While one can argue that most of these tweaks are pretty easy to implement, and there is a vast number of educational material available,

the downside comes very clear when working with models that do not even fit on a single GPU. These models have to be trained in a distributed manner across multiple devices. When doing so, the training loop itself gets much more complex and challenging to implement.

CHAPTER

THREE

EXPERIMENTAL DESIGN

3.1 Task

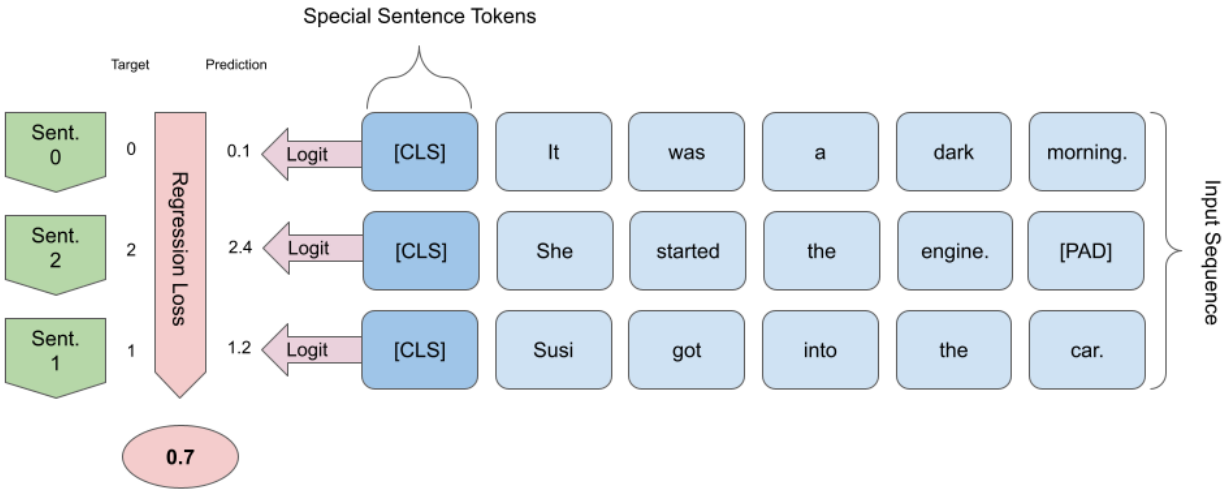

Fig. 3.1: Visualization of the sentence ordering task.

To compare the frameworks, we will implement the same experiment with each of them. The task of the experiment is a critical choice since training a model on a standard objective like text- or token classification would not require much customization. Also, it would put the Huggingface Trainer into an advantageous position because it supports such tasks out of the box. To ensure a fair comparison, we chose another quite exotic objective: Sentence Ordering. Our goal is to train a model to predict the correct order of a sequence of shuffled sentences. This task seemed right for two reasons. Firstly, it can be implemented with a standard Huggingface model but requires a custom loss function. Secondly, the task falls into the category of self-supervised learning. So it is possible to generate training and test data from unstructured text data in an effortless manner. Besides these practical implications, the objective is interesting because it can measure the causal coherence of texts. For example, to assess whether the coherence of actions varies between different text types or genres.

But how do we achieve this task? There are various methods proposed ranging from relatively simple approaches like applying ranking-loss functions (Zhu *et al.* [2021]) to rather complex ones that learn a graph representation of the sentences

and then use topological sorting to extract the correct order of the sentences (Yin *et al.* [2019]). Because we do not care much about achieving state-of-the-art results, we opt for one of the most straightforward approaches and frame it as a regression task. Fig. 3.1 visualizes this approach. The model should output a regression score for each sentence in the input, indicating its position in the original text. Therefore, a special token is added as a prefix to each sentence. These tokens act as our targets while training, and they should output a value near to the original index of the sentence in the correct ordered text. The loss is measured using the Mean-Squared-Error objective The target value for each sentence token is not normalized and ranges from 0 to N, where N is the number of sentences in the input sequence. Another even more straightforward approach would be to add a final layer to the network with a fixed size of neurons (one for each sentence), but this would mean we had to know the number of sentences in the input beforehand, which would harm the usability of the model.

Using regression for sentence ordering is not a novel approach. It was first proposed by McClure *et al.* [2018], who used it with CNNs and LSTMs and was later on employed as a baseline by Kumar *et al.* [2020] with BERT. However, in both cases, the authors used the neural network to encode all sentences independently and then fed the sentence embeddings into a regression component. But we will feed the whole shuffled text into the network. This strategy allows the model to attend to all sentences and their tokens simultaneously, giving it more context to decide on the correct order. Also, both other authors normalized the values of the regression target to be in the range of $-1, 1$ or $0, 1$, respectively. However, after some brief experiments, we dropped the normalization because it did not yield any benefits. But, in general, dropping the normalization is only feasible with a dataset with a fixed number of sentences because otherwise, it could skew the loss in favor of short texts.

Since the position of the target tokens in the input sequences differs, we need our language model to output one logit for each token. Two Huggingface model variants return a suitable output `<...ModelType...>ForTokenClassification` or `<...ModelType...>ForQuestionAnswering`. We chose the first one, but all the code in the following section should also run when employing a model with a question-answering head.

3.2 Metrics

To measure the performance of our model, we use two metrics.

Accuracy

Accuracy measures how many sentences per instance are indexed correctly. Accuracy gives a rough estimate of how well the model performs, but it can paint a misleading picture since it does not fully account for our task's ranking aspect. For example, suppose that the model would correctly predict that sentence B follows sentence A and expects them to be at position 0 and 1 in the total ordering. But in reality, they are the last sentences of the text. So, in this case, the accuracy would be 0 (assuming that all other predictions were also wrong).

Kendalls Tau

In contrast to accuracy, Kendall Tau is a ranking correlation coefficient that accounts for partially correct parts of a ranking. It measures the difference between pairs of sentences correctly predicted as following and all other wrongly predicted pairs. This value is further corrected for the chance of randomly predicting correct pairs of sentences by dividing it by the total number of unique ways to pick two sentences from the sequence.

$$\tau_{\text{Kendall}} = \frac{\#\text{Correctly predicted pairs of sentences} - \#\text{Wrongly predicted pairs of sentences}}{\binom{N}{2}}$$

3.3 Dataset

We use the 2017 version of the ROCStories dataset by Mostafazadeh *et al.* [2016]. It contains 52.665 short stories with a fixed length of five sentences. This dataset is commonly used in the literature because its stories mainly depict concrete actions with a clear causal order, making them relatively simple to understand without leaving much space for ambiguities. This property makes it a good fit for testing the general capabilities of language models on this task.

Warning: Even though the ROCStories dataset is freely available to the public, anyone who uses it must submit contact data. So the dataset itself is not included in the Github-Repository and must be downloaded independently from https://cs.rochester.edu/nlp/rocstories/

Note: In addition, we tested the same experimental setup on a dataset of sentences sampled from german short novels (Novellen) without much success. An insufficient sampling of subparts of the texts is the most likely reason for this failure. Applying this task to all kinds of different textual domains can be a fruitful question itself but lies outside the scope of this work.

CHAPTER

FOUR

PREREQUISITES

The following experiments share the same general logic, but the concrete implementation will differ in minor details since each framework has another structural approach. So before we start, we will take a brief look at the general logic for the data loading parts of the experiment and the computation of the loss function.

4.1 Dataset-preparation

To load the stories, shuffle the sentences, and further prepare, we use Huggingface's Datasets library, which provides various useful functions for manipulating text data. Because Huggingface Datasets are fully compatible with PyTorch's class for data-loading, they can also be used by all non-Huggingface libraries without further adjustments.

The preparation itself is simple:

```
from datasets import Dataset, DatasetDict
```

At first, we load the dataset in its original format.

```
dataset = Dataset.from_csv('../scripts/data/ROCStories_winter2017 - ROCStories_
↪winter2017.csv')
dataset
```

```
Dataset({
    features: ['storyid', 'storytitle', 'sentence1', 'sentence2', 'sentence3',
↪'sentence4', 'sentence5'],
    num_rows: 52665
})
```

We got 52.665 stories. Each one has a length of five sentences. Additionally, each text has a short title, but we discard them.

```
len(dataset)
```

```
52665
```

```
print(dataset[0])
```

```
{'sentence1': 'David noticed he had put on a lot of weight recently.',
 'sentence2': 'He examined his habits to try and figure out the reason.',
```

(continues on next page)

(continued from previous page)

```
 'sentence3': "He realized he'd been eating too much fast food lately.",
 'sentence4': 'He stopped going to burger places and started a vegetarian '
              'diet.',
 'sentence5': 'After a few weeks, he started to feel much better.',
 'storyid': '8bbe6d11-1e2e-413c-bf81-eaea05f4f1bd',
 'storytitle': 'David Drops the Weight'}
```

Next, we create the training data by shuffling the sentences and creating labels indicating the original order. Also, we add special tokens to each sentence.

We implement the shuffling process using the `.map`-method of the `Dataset`-class. Following the library's out-of-place policy, the `.map`-method returns a new dataset containing the changes instead of changing the dataset it was called on.

The `.map`-method has two modes: batch-mode or single entry mode. Either way, it receives a dictionary as input where each key represents a column of the dataset. In single entry mode, the values of the input dictionary hold one entry in the dataset. In batch mode, the values are lists containing more than one entry. The following function only works in both modes since it converts both input formats to the same intermediate form, but in general, the batch mode should be preferred to save time. The output of the function has to be a dictionary in the same format as the input.

```
from random import shuffle
from random import seed as set_seed

def make_shuffle_func(sep_token):
    def shuffle_stories(entries, seed=42):
        set_seed(seed)
        entries_as_dicts = [
            dict(zip(entries, values))
            for values in zip(*entries.values())
        ]
        converted_entries = []
        for entry in entries_as_dicts:
            sents = [
                entry[key]
                for key in sorted(
                    [key for key in entry.keys() if key.startswith('sentence')
                    ], key=lambda x: int(x[-1])
                )
            ]
            sent_idx = list(range(len(sents)))
            sents_with_idx = list(zip(sents, sent_idx))
            shuffle(sents_with_idx)
            text = f'{sep_token} ' + f' {sep_token} '.join(
                [s[0]for s in sents_with_idx]
            )
            so_targets = [s[1] for s in sents_with_idx]
            shuffled_entry = {'text': text, 'so_targets': so_targets}
            converted_entries.append(shuffled_entry)
        new_entry = {
            key: [entry[key] for entry in converted_entries]
            for key in converted_entries[0]
        }
        return new_entry
    return shuffle_stories
```

`[CLS]` is one of the special tokens of models directly descending from BERT. During the pretraining stage, it learns a representation of the whole input sequence and thus only occurs once in each input. Since we do not need a representation of the input as a whole, we use it as the special sentence token.

```
map_func = make_shuffle_func('[CLS]')
```

```
dataset = dataset.map(map_func, batched=True)
```

```
  0%|          | 0/53 [00:00<?, ?ba/s]
```

After applying the shuffle function, the dataset has two additional columns. The `text` column contains the shuffled and concatenated sentences, and the `so_targets` column contains the indices of the sentences in the original order. For example, in the first text in the dataset, the first sentence in the shuffled text is 4th place in the original order.

```
print(dataset[0])
```

```
{'sentence1': 'David noticed he had put on a lot of weight recently.',
 'sentence2': 'He examined his habits to try and figure out the reason.',
 'sentence3': "He realized he'd been eating too much fast food lately.",
 'sentence4': 'He stopped going to burger places and started a vegetarian '
              'diet.',
 'sentence5': 'After a few weeks, he started to feel much better.',
 'so_targets': [3, 1, 2, 4, 0],
 'storyid': '8bbe6d11-1e2e-413c-bf81-eaea05f4f1bd',
 'storytitle': 'David Drops the Weight',
 'text': '[CLS] He stopped going to burger places and started a vegetarian '
         'diet. [CLS] He examined his habits to try and figure out the reason. '
         "[CLS] He realized he'd been eating too much fast food lately. [CLS] "
         'After a few weeks, he started to feel much better. [CLS] David '
         'noticed he had put on a lot of weight recently.'}
```

Lastly, we want to split our dataset into three subsets. The train-set is used for training. The validation set can be used to validate the performance during training or hyperparameter optimization. The test set will be used for the final evaluation of the final model.

```
train_test = dataset.train_test_split(test_size=0.2, seed=42)

test_validation = train_test['test'].train_test_split(test_size=0.3, seed=42)

dataset = DatasetDict({
    'train': train_test['train'],
    'test': test_validation['train'],
    'val': test_validation['test']})
dataset
```

```
DatasetDict({
    train: Dataset({
        features: ['sentence1', 'sentence2', 'sentence3', 'sentence4', 'sentence5',
 ↪ 'so_targets', 'storyid', 'storytitle', 'text'],
        num_rows: 42132
    })
    test: Dataset({
        features: ['sentence1', 'sentence2', 'sentence3', 'sentence4', 'sentence5',
 ↪ 'so_targets', 'storyid', 'storytitle', 'text'],
        num_rows: 7373
    })
    val: Dataset({
```

(continues on next page)

(continued from previous page)

```
        features: ['sentence1', 'sentence2', 'sentence3', 'sentence4', 'sentence5',
↪ 'so_targets', 'storyid', 'storytitle', 'text'],
        num_rows: 3160
    })
})
```

Finally, we save the dataset.

```
dataset.save_to_disk('rocstories')
```

4.2 Data loading

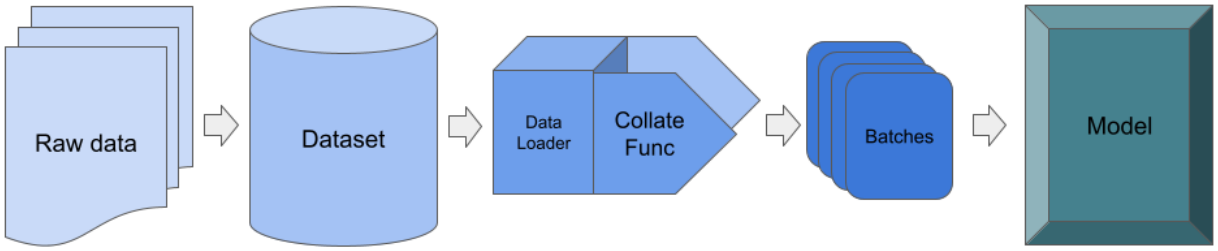

Fig. 4.1: High-level visualization of data-flow from the original dataset to the model.

From a high-level view, a Huggingface `Dataset` can be seen as a table with columns that correspond to attributes (called features) and rows representing one dataset entry. From a more concrete technical perspective, the `Dataset`-instance provides an iterable that yields a dictionary for each entry in the dataset. Each dictionary contains attribute-value pairs.

```
from datasets import load_from_disk

dataset = load_from_disk('../scripts/data/rocstories')
print(dataset['train'].features)
```

```
{'sentence1': Value(dtype='string', id=None),
 'sentence2': Value(dtype='string', id=None),
 'sentence3': Value(dtype='string', id=None),
```

(continues on next page)

(continued from previous page)

```
 'sentence4': Value(dtype='string', id=None),
 'sentence5': Value(dtype='string', id=None),
 'so_targets': Sequence(feature=Value(dtype='int64', id=None), length=-1, id=None),
 'storyid': Value(dtype='string', id=None),
 'storytitle': Value(dtype='string', id=None),
 'text': Value(dtype='string', id=None)}
```

```
print(dataset['train'][0])
```

```
{'sentence1': 'Bob took his daughter canoeing on the river.',
 'sentence2': 'The entire trip was about three miles.',
 'sentence3': 'The current was very fast through the shoals.',
 'sentence4': 'They got hung up on a rock near the end.',
 'sentence5': 'They had a good time and pledged to go again.',
 'so_targets': [3, 4, 2, 0, 1],
 'storyid': '2696704e-60a8-44de-99ea-bed2dada2a68',
 'storytitle': 'Canoeing',
 'text': '[CLS] They got hung up on a rock near the end. [CLS] They had a good '
         'time and pledged to go again. [CLS] The current was very fast '
         'through the shoals. [CLS] Bob took his daughter canoeing on the '
         'river. [CLS] The entire trip was about three miles.'}
```

We can't feed the model with raw texts, so we have to tokenize them beforehand.

As stated before, each model comes with a custom tokenizer, so we have to load it, just like the model itself.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('bert-base-cased', return_dict=True)
tokenized_text = tokenizer("Jimmy went down the road.")
print(tokenized_text)
```

```
{'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1],
 'input_ids': [101, 4479, 1355, 1205, 1103, 1812, 119, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0]}
```

The tokenizer takes a text or a collection of texts and converts it to a tokenized sequence. Also, it creates additional inputs for the model, such as the attention mask.

To tokenize the whole dataset, we can once again use the map function.

```
def make_tokenization_func(tokenizer, text_column, *args, **kwargs):
    def tokenization(entry):
        return tokenizer(entry[text_column], *args, **kwargs)
    return tokenization

tokenization = make_tokenization_func(
    tokenizer=tokenizer,
    text_column="text",
    padding="max_length",
    truncation=True,
    add_special_tokens=False,
    return_tensors='np'
)
```

(continues on next page)

(continued from previous page)

```
dataset = dataset.map(tokenization, batched=True)
print(dataset['train'][0].keys())
```

```
  0%|          | 0/43 [00:00<?, ?ba/s]
```

```
  0%|          | 0/8 [00:00<?, ?ba/s]
```

```
  0%|          | 0/4 [00:00<?, ?ba/s]
```

```
dict_keys(['attention_mask', 'input_ids', 'sentence1', 'sentence2', 'sentence3',
 ↪'sentence4', 'sentence5', 'so_targets', 'storyid', 'storytitle', 'text', 'token_
 ↪type_ids'])
```

To feed the data to the neural network, we have to split it up into batches of a fixed size. To do so, PyTorch provides a general class, called `torch.utils.data.DataLoader`, that takes in iterable and returns batches just in time while training.

The `DataLoader` class is agnostic towards the data it receives. To create batches that are compatible with the Huggingface model, we have to pass it a function that takes in multiple entries from our dataset and converts them into the correct format.

This function is called `collate_fn` and can be specified while initiating the `DataLoader` object. Using a simple identity function, we see that the `collate_fn` receives a list with B entries where B is the batch size.

```
from torch.utils.data import DataLoader

def identity(batch):
    return batch

data_loader = DataLoader(dataset['train'], batch_size=2, collate_fn=identity)
batch = next(iter(data_loader))
print(len(batch))
print(type(batch))
print(batch[0].keys())
```

```
2
<class 'list'>
dict_keys(['attention_mask', 'input_ids', 'sentence1', 'sentence2', 'sentence3',
 ↪'sentence4', 'sentence5', 'so_targets', 'storyid', 'storytitle', 'text', 'token_
 ↪type_ids'])
```

Huggingface provides a collate function that can convert tokenized data into batches in a suitable format. The Huggingface collation function only works with numeric data such as scalars or arrays. So we have to drop all texts before we pass the dataset into the `Dataloader` object.

```
dataset = dataset.remove_columns(
    ["text", "storyid", "storytitle"] + [f"sentence{i}" for i in range(1, 6)]
)
dataset.set_format("torch")
print(dataset["train"].features)
```

```
{'attention_mask': Sequence(feature=Value(dtype='int8', id=None), length=-1,␣
↪id=None),
 'input_ids': Sequence(feature=Value(dtype='int32', id=None), length=-1, id=None),
 'so_targets': Sequence(feature=Value(dtype='int64', id=None), length=-1, id=None),
 'token_type_ids': Sequence(feature=Value(dtype='int8', id=None), length=-1,␣
↪id=None)}
```

After only numeric data is left, we have to face the last problem in the collation problem. The Huggingface collation function only handles arrays of the same shape when collating them into one batch. In theory (e.g., with other datasets), we could have a varying number of labels if we wanted to work shuffled texts with a variable number of sentences. We tackle this problem by introducing a custom collation function to make our preparation pipeline as flexible as possible.

```
from transformers import default_data_collator
from torch.nn.utils.rnn import pad_sequence

def so_data_collator(batch_entries, label_key='so_targets'):
    """
    Custom dataloader to apply padding to the labels.
    """
    label_dicts = []

    # We split the labels from the rest to process them independently
    for entry in batch_entries:
        label_dict = {}
        for key in list(entry.keys()):
            if label_key in key:
                label_dict[key] = entry.pop(key)
        label_dicts.append(label_dict)

    # Everything except our labels can easily be handled by the "default collator"
    batch = default_data_collator(batch_entries)

    # We need to pad the labels 'manually'
    for label in label_dicts[0]:
        labels = pad_sequence(
            [label_dict[label] for label_dict in label_dicts],
            batch_first=True,
            padding_value=-100,
        )

        batch[label] = labels
    return batch
```

This function used the Huggingface default collation function to handle everything except the labels. The labels are padded with a batch-wise max length strategy and added to the batch.

```
data_loader = DataLoader(dataset['train'], batch_size=2, collate_fn=so_data_collator)
batch = next(iter(data_loader))
print(batch)
```

```
{'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]]),
 'input_ids': tensor([[ 101, 1220, 1400,  ...,    0,    0,    0],
        [ 101, 1109, 3336,  ...,    0,    0,    0]]),
 'so_targets': tensor([[3, 4, 2, 0, 1],
```

(continues on next page)

(continued from previous page)

```
        [1, 0, 2, 3, 4]]),
 'token_type_ids': tensor([[0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0]])}
```

Now the data is in the correct format for training.

4.3 Loss function

As stated in the experimental design, we use a plain Mean-Squared-Error regression loss. Since we only want to consider the special tokens, we must select them before the actual computation. Therefore, we need the `input_ids` to figure out their position in the sequence.

To compute the loss for one single batch, we add the loss scores of all sentences of one text and take the average of all batch entries. Due to computational constraints, transformer-based language models typically have a limit on the input size. So our inputs might have to be truncated to fit into the model. In this case, we discard the labels for the sentences left out and only consider the data that fits into the model.

The following listing contains a general implementation of the loss function:

```
import torch
from torch import nn

def sentence_ordering_loss(batch_logits, batch_targets, batch_input_ids, target_token_
↪id) -> torch.Tensor:
    # Since we have varying number of labels per instance, we need to compute the␣
↪loss manually for each one.
    loss_fn = nn.MSELoss(reduction="sum")
    batch_loss = torch.tensor(0.0, dtype=torch.float64, requires_grad=True)
    for labels, logits, input_ids in zip(
        batch_labels, batch_logits, batch_input_ids
    ):
        # Firstly, we need to convert the sentence indices to regression targets.
        # Also we need to remove the padding entries (-100)
        true_labels = labels[labels != -100].reshape(-1)
        targets = true_labels.float()

        # Secondly, we need to get the logits from each target token in the input␣
↪sequence
        target_logits = logits[input_ids == target_token_id].reshape(-1)

        # Sometimes, we will have less target_logits than targets due to trunction of␣
↪the input.
        # In this case, we just consider as many targets as we have logits
        if target_logits.size(0) < targets.size(0):
            targets = targets[: target_logits.size(0)]

        # Finally we compute the loss for the current instance and add it to the␣
↪batch loss
        batch_loss = batch_loss + loss_fn(targets, target_logits)

    # The final loss is obtained by averaging over the number of instances per batch
    loss = batch_loss / batch_logits.size(0)

    return loss
```

CHAPTER

FIVE

HUGGINGFACE TRAINER

Since Huggingface proclaimed goal is to provide an environment to develop and train all sorts of language models, they also ship a solution for training models. It is called the `Trainer`, and is integrated into the `transformers` library itself. Of course, it is profoundly integrated into the Huggingface-ecosystem and can train most `transformers` models out of the box.

5.1 Classes

5.1.1 Trainer

Design-wise, the `Trainer` is one single class that handles the training end-to-end. Its configuration is outsourced to a `TrainingArguments` class that stores all relevant parameters for training. These arguments are passed to the `Trainer` alongside a model and a dataset during initialization. Since Huggingface models compute the loss internally, the `Trainer` passes the input data to the model, extracts the loss from the output, and does the backward step. It also handles additional steps to monitor the training process, like saving checkpoints of the model or logging the loss and other validation metrics. A significant advantage of using the `Trainer` is its ability to do multi-device training without requiring the user to care about dispatching the models and data to multiple accelerators. Also, it comes with an extension that allows more sophisticated tweaks, like training with 16bit-precision.

Extending the `Trainer`

There are two different options to customize certain aspects of the behavior of the `Trainer`. Additional read-only operations can be implemented with the callback API. Callbacks are executed at specific events during the training (e.g., at the end of an epoch). They have access to many different things like the model or the current state of the `Trainer`. However, since they can not manipulate their environment, their scope is limited to logging, saving certain parts of a model, or stopping the training if a specific condition is met.

If further changes to the `Trainer` are required, the recommended way is to subclass it and create a custom via inheritance. Internally, the `Trainer` structures the training into different sub-steps and exposes them via a method for each of them. By overwriting these methods, it is possible to change certain parts of the logic without rewriting the rest of the code that would not be changed anyway. The most important methods to modify the `train-test-val`-loop itself are the `<train/test/val>-step` methods and the `compute_loss` method. These methods implement the essential individual training steps and are called within methods that implement higher-order operations like the `.train`-method, which handles the complete training loop.

Logging

If a `logdir`-argument is specified in the `TrainingArguments`-object, logging is enabled automatically. By default, the `Trainer` outputs the logs in two formats: Stdout and disk, using a Tensorboard-compliant format. Additional logging can be implemented by either overwriting the `.log`-method of Trainer or by using callbacks. There are already some pre-built callbacks available. For example, to log the progress to Weights and Biases or a CSV table.

Custom metrics

Since the `Trainer` is agnostic towards the task it is used with; it only logs the loss by default. Additionally, metrics can be added by equipping the `Trainer` with a function that computes them during initialization. This function receives an `EvalPrediction` object. This object holds all predictions of the model and the valid labels. The output of the custom metric function ought to be a dictionary containing the name of the metric as key and the score as value.

5.1.2 Training Arguments

As stated above, a `TrainingArguments` object stores all hyperparameters of the training. Storing all parameters in a single object is helpful to ensuring reproducibility since this object can easily be serialized and saved to disk as JSON using its `.to_json_string`-method. Also, the `TrainingArguments` class works seamlessly with the built-in CLI-parser class of `transformers`, which helps make the configuration of an experiment available through a command-line interface.

5.1.3 HfArgumentParser

Most experiments are repeated several times with different parameters. These parameters have to be changed directly in the source code by default, which is not ideal for several reasons. Most importantly, it can harm reproducibility since tracking changes in the source code requires either version control and a strict commit regime or keeping several versions of the same file with different parameters. Also, it can be tedious the search for the location of all parameters across the code manually. Making the hyperparameters adjustable via a command-line interface decouples their configuration from the rest of the code, alleviating this issue. While there are arguably a lot of different solutions to this problem with many strategies that are more sophisticated than a command-line interface, it is an excellent first step. Moreover, it has the advantage of being platform-independent without depending on additional dependencies.

Huggingface provides a built-in solution for building these interfaces called `HfArgumentParser`. It is an extended version of Pythons `argsparse` parser and creates command-line interfaces by parsing the fields of `dataclasses` and exposing them as command-line arguments. Since most configuration classes of the `transformers` library are `dataclasses`, the `HfArgumentParser` can flexibly control nearly every aspect of the training. Further extending the arguments can be easily done by creating custom `dataclasses` that hold additional parameters.

```
from dataclasses import dataclass, field
from transformers import HfArgumentParser

@dataclass
class TrainArgs:
    batch_size: int = field(
        default = 8,
        metadata = {"help": "Number of batched for training."}
    )

parser = HfArgumentParser(TrainArgs)
parser.print_help()
```

(continues on next page)

(continued from previous page)

```
train_args = parser.parse_args_into_dataclasses(["--batch_size", "4"])
print(train_args)
```

```
usage: ipykernel_launcher.py [-h] [--batch_size BATCH_SIZE]

optional arguments:
  -h, --help            show this help message and exit
  --batch_size BATCH_SIZE
                        Number of batched for training. (default: 8)
(TrainArgs(batch_size=4),)
```

5.2 Implementation

5.2.1 Loss function

For the sentence ordering task, we employ a language model with a standard token-classification head. However, since the task requires a custom loss function, we have to discard the loss of the model and use our custom loss function. To do so, we follow the guidelines and create our custom version of the `Trainer` with a custom `.compute_loss` function. The implementation is straightforward. The `.compute_loss` method receives a reference to the model and the input data as inputs, which is especially helpful in cases like ours where we need to check the `input_ids` to compute the loss. In addition, to our custom loss function, we also add another attribute to the `Trainer`, which holds the id of the target sentence token in order to find the correct tokens in the input sequence. We leave the rest of the `Trainer` untouched.

```
class SentenceOrderingTrainer(Trainer):
    def __init__(self, *args, **kwargs):
        self.target_token_id = kwargs.pop("target_token_id")
        super().__init__(*args, **kwargs)

    def compute_loss(self, model, inputs, return_outputs=False):

        # Get sentence indices
        batch_labels = inputs.pop("labels")

        # Get logits from model
        outputs = model(**inputs)
        batch_logits = outputs["logits"]

        # Get logits for all cls tokens
        batch_input_ids = inputs["input_ids"]

        loss_fn = nn.MSELoss(reduction="sum")
        batch_loss = torch.tensor(0.0, dtype=torch.float64, requires_grad=True)

        for labels, logits, input_ids in zip(
            batch_labels, batch_logits, batch_input_ids
        ):

            true_labels = labels[labels != -100].reshape(-1)
            targets = true_labels.float()

            target_logits = logits[input_ids == self.target_token_id].reshape(-1)
```

(continues on next page)

(continued from previous page)

```
            if target_logits.size(0) < targets.size(0):
                targets = targets[: target_logits.size(0)]

            batch_loss = batch_loss + loss_fn(targets, target_logits)

        loss = batch_loss / batch_logits.size(0)

        outputs["loss"] = loss
        return (loss, outputs) if return_outputs else loss
```

5.2.2 Metrics

To compute custom metrics during validation, we need to create a function. The function computes all metrics at once. In contrast to the `.compute_loss`-method, which receives the input and the model, it receives an `EvalPrediction` object as input. An `EvalPrediction` contains the model's outputs and the labels from the dataset. However, similar to the loss function, computing the metrics requires access to the input data to retrieve the indices of the target tokens. To control the content of an `EvalPrediction` object, we can use the `label_names` parameter of the `TrainingArguments`. With this argument, we can specify additional fields that are copied from the input batches to the `EvalPrediction` objects. This way, we can incorporate the labels and the `input_ids` of tokens in the `EvalPrediction` object.

A minor but valuable trait of the `EvalPrediction` objects is that their content gets converted from `torch.tensors` to `np.arrays`. Because most validation metrics from other libraries use NumPy, we do not need to convert the data manually.

```
training_args = TrainingArguments(
    ...,
    label_names=["labels", "input_ids"],
    ...,
)
```

```
def make_compute_metrics_func(target_token_id) -> Callable:
    def compute_ranking_func(eval_prediction: EvalPrediction) -> Dict[str, float]:
        batch_sent_idx, batch_input_ids = eval_prediction.label_ids
        batch_logits = eval_prediction.predictions.squeeze(2)

        metrics = defaultdict(list)
        for sent_idx, input_ids, logits in zip(
            batch_sent_idx, batch_input_ids, batch_logits
        ):
            sent_idx = sent_idx.reshape(-1)
            input_ids = input_ids.reshape(-1)
            logits = logits.reshape(-1)

            sent_idx = sent_idx[sent_idx != 100]
            target_logits = logits[input_ids == target_token_id]
            if sent_idx.shape[0] > target_logits.shape[0]:
                sent_idx = sent_idx[: target_logits.shape[0]]
            predicted_idx = np.argsort(np.argsort(target_logits))
            tau, pvalue = kendalltau(sent_idx, predicted_idx)
            metrics["kendalls_tau"].append(tau)
            metrics["acc"].append(accuracy_score(sent_idx, predicted_idx))
```

(continues on next page)

(continued from previous page)

```
            metrics["mean_logits"].append(logits.mean())
            metrics["std_logits"].append(logits.std())
        metrics = {metric: np.mean(scores) for metric, scores in metrics.items()}
        return metrics

    return compute_ranking_func
```

5.2.3 Custom CLI arguments

We use the `HfArgumentParser` to make the parameters of our experiment adjustable via the command line. In addition to the `TrainingsArguments`, we also want to control the type of the model. Custom parameters can easily be added by creating a custom `dataclass`. We create a `ModelArgs` class that has two fields. One to specify the name or path to the model and a second parameter to specify the path where the final model is saved after training.

```
from dataclasses import dataclass, field
from transformers import TrainingArguments, HfArgumentParser

@dataclass
class ModelArgs:
    model_name_or_path: str = field(
        default="bert-base-cased",
        metadata={
            "help": "Path to pretrained model or model or its name to load it from␣
 ↪Huggingface Hub."
        },
    )

    final_checkpoint_path: str = field(
        default=None, metadata={"help": "Path to save the final model."}
    )

...

args_parser = HfArgumentParser((ModelArgs, TrainingArguments))
model_args, training_args = args_parser.parse_args_into_dataclasses()
```

5.3 Complete code

After moving our custom code for the `Trainer` and the metric function to an external module, the rest of the code to implement the experiment looks like Listing (TODO). There are only two steps left to complete the script. Firstly, we must ensure that our data always contains the correct special tokens for ordering the sentences. Since we prepared the data beforehand by adding BERTs special `[SEP]`-token as a prefix to each sentence, we have to ensure that these tokens are replaced if necessary using the `replace_cls_token` function.

Lastly, we want to control the randomness in our experiment to make it consistently reproducible. The `transformers` library comes with a helpful function called `set_seed`, which controls the state of all random number generators of Python itself, NumPy, and PyTorch at once.

```
import json
from transformers import TrainingArguments, HfArgumentParser
from transformers import AutoModelForTokenClassification, AutoConfig, AutoTokenizer
```

(continues on next page)

(continued from previous page)

```
from transformers import set_seed
from datasets import load_from_disk

from model import (
    SentenceOrderingTrainer,
    so_data_collator,
    make_compute_metrics_func,
    ModelArgs,
    make_tokenization_func,
)


if __name__ == "__main__":

    args_parser = HfArgumentParser((ModelArgs, TrainingArguments))
    model_args, training_args = args_parser.parse_args_into_dataclasses()

    # Add fixed args
    training_args.label_names = ["labels", "input_ids"]

    set_seed(training_args.seed)

    dataset = load_from_disk(
        "/home/keller/Uni/trf_training_tut/scripts/data/rocstories"
    )

    tokenizer = AutoTokenizer.from_pretrained(model_args.model_name_or_path)

    if tokenizer.cls_token != "[CLS]":
        print(
            f"Model does not a have a [CLS] token. Updating the data with token
 ↪{tokenizer.cls_token} ..."
        )

        def replace_cls_token(entry):
            texts = entry["text"]
            replaced_texts = []
            for text in texts:
                replaced_texts.append(text.replace("[CLS]", tokenizer.cls_token))
            entry["text"] = replaced_texts
            return entry

        dataset = dataset.map(replace_cls_token, batched=True)

    model_config = AutoConfig.from_pretrained(
        model_args.model_name_or_path, num_labels=1
    )
    model = AutoModelForTokenClassification.from_pretrained(
        model_args.model_name_or_path, config=model_config
    )

    tokenization = make_tokenization_func(
        tokenizer=tokenizer,
        text_column="text",
        padding="max_length",
        truncation=True,
```

(continues on next page)

(continued from previous page)

```
        add_special_tokens=False,
    )
    dataset = dataset.map(tokenization, batched=True)

    dataset = dataset.rename_column("so_targets", "labels")

    dataset.set_format("torch")

    metrics_func = make_compute_metrics_func(tokenizer.cls_token_id)

    trainer = SentenceOrderingTrainer(
        model=model,
        args=training_args,
        train_dataset=dataset["train"],
        eval_dataset=dataset["val"],
        target_token_id=tokenizer.cls_token_id,
        data_collator=so_data_collator,
        compute_metrics=metrics_func,
    )

    trainer.train()

    trainer.save_model(model_args.final_checkpoint_path)

    test_results = trainer.evaluate(eval_dataset=dataset["test"])
    with open(f"test_results_{model_args.model_name_or_path}.json", "w") as f:
        json.dump(test_results, f)

    print(test_results)
```

5.4 Conclusion

The Huggingface `Trainer` is a perfect choice when training models on standard tasks that are well supported. In these cases, it enables to train models effortlessly without requiring to write much code. In the best case, when the dataset is already available as Huggingface `Dataset`, it comes down to a few lines of code to train the model without having to dive deep into any internals along the way.

Also, it has many useful out-of-the-box features, like gradient clipping, half-precision training, support of distributed training, or logging to Tensorboard, which make it feasible for training large models on large datasets.

Nonetheless, there are a few issues if one wants to leave the carved-out paths. Like the rest of Huggingface's software, the `transformers` library is relatively new and evolves at great speed. Huggingface's self-proclaimed goal is to provide an easy-to-use all-in-one infrastructure for NLP with language models and incorporate new models, architectures, and developments as quickly as possible. On this path, sacrifices have to be made.

One area that seems to suffer from the speedy development is documentation. It is sufficient and provides all essential information, but it can sometimes be very sparse in detail. Often, there are multiple options to choose from when customizing something. For example, the default optimizer can be exchanged during the initialization of the `Trainer`, by simply passing another one to it. Or by overwriting the `.create_optimizer`-method. In cases like this one, the documentation lacks hints to decide which way to go.

Other times the documentation does not paint the whole picture of the behavior of the described object. In these cases, it might become necessary to take a look into the source code itself.

By looking into the source code of `Trainer`, it becomes evident that it could use some refactoring. Especially, its high-level methods, like the `.train`-method, are very complex since they do much heavy lifting, for example, dispatching the training to multiple devices. While the preferred way to customize the training is to subclass the `Trainer` and overwrite methods, this is only feasible for the low-level methods that define single steps. Even tiny adjustments to the high-level methods can require copying code or rewriting certain parts.

CHAPTER

SIX

PYTORCH LIGHTNING

In contrast to the Huggingface `Trainer`, which handles the complete training itself, PyTorch Lightning (Falcon [2019]) takes a different approach. It not only aims at handling the training but also at structuring the creation of a model too. Its main goal is not to hide complexity from the user but to provide a well-structured API for building neural networks of all kinds. The most striking aspect of this is that in PyTorch Lightning's philosophy, a model and its inference-, training and prediction logic are not separate things that can be exchanged independently. Instead, it binds all these parts directly to the model itself. In doing so, PyTorch-Lightning does not make any assumptions on the nature of the model or the training itself. Thus it allows covering many tasks and domains with maximum flexibility.

However, this approach comes at the cost that the user again must implement many things manually. Naturally, this approach is keener to researchers who implement and test custom models, while practitioners who only want to employ pre-built models must deal with some implementational overhead. PyTorch Lightning's steep learning curve compounds this issue. However, there is exhaustive documentation with many tutorials (as texts and videos), best practices, and user guides on building various models across different domains. Also, if an experiment is implemented in PyTorch Lightning, there are a lot of helpful tweaks and techniques to improve or speed up the training. So that it can be worthwhile even when using pre-built models. These facilitation features include tweaks like training with half-precision, automatic tuning of the learning rate, and integrations into hyperparameter tuning frameworks or creating command-line interfaces to control the parameters. In addition to that, there is support for different computational backends that help to dispatch the training on multiple accelerators like GPUs and TPUs. If these features are not enough, there is a growing ecosystem of third-party extensions, widening the scope and functionality of the framework.

6.1 Classes

From a technical point of view, Pytorch Lightning provides an API composed of three different main classes, dividing the training process into a sequence of single atomic steps. These classes implement the model, the logic for storing and processing the training data, and the training process itself.

6.1.1 `LightningModule`

A subclass of a `LightningModule` implements the model. A `LightningModule` is an extended version of PyTorch's `nn.Module` class. `nn.Modules` are the basic building blocks of neural networks in PyTorch. In essence, they store a set of parameters, for example, weights of a single layer alongside with `.forward`-method that defines the computational logic when data flows through the module. They are designed to work recursively. One module can be composed of several submodules so that each building block of a neural network, starting from single layers up to a complete network, can be implemented in this one class. The following listing shows an exemplary implementation of a simple linear layer as `nn.Module`. By chaining multiple instances of the dense layer in a `nn. Sequential` class, it is possible to create a simple feed-forward network. This network is again a subclass of the `nn.Module` class.

```
import torch
from torch import nn

class DenseLayer(nn.Module):
    """Fully connected linear layer."""

    def __init__(self, in_shape, out_shape):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(in_shape, out_shape), requires_
 ↪grad=True)

    def forward(self, inputs):
        return torch.matmul(inputs, self.weights)

network = nn.Sequential(
    DenseLayer(512, 16),
    nn.ReLU(),
    DenseLayer(16, 8),
    nn.ReLU(),
    DenseLayer(8, 2)
)

inputs = torch.randn(8, 512)  # Batchsize 8
outputs = network(inputs)
print(outputs.size())
print(issubclass(nn.Sequential, nn.Module))
```

```
torch.Size([8, 2])
True
```

A `LightningModule` is intended to replace the outmost `nn.Module` instance of a model, which holds the complete network. It extends the `nn.Module` class with new methods, designed to structure not only the logic of a single forward pass but also other steps like a complete train-, test- or validation-steps. With this extension, it becomes possible to define a single forward step through the network and how the models should be trained and tested as well. In essence, it provides a way to incorporate the training loop into the model itself. This strategy has one massive advantage over the standard PyTorch practice of writing an external function that implements the training loop. It helps to make the model self-contained, meaning that it holds all necessary logic itself. This property alleviates sharing models since only one class carries all information to train and test the model.

Training, Testing, Validation

The methods for training, testing and prediction are called `.train_step`, `.validation_step` and `.test_step` respectively. They all define how a single batch of data should be handled for these steps. Typically the `.train_step`-method computes the loss score, and the other two methods compute other validation metrics. Design-wise, only the `.train_step`-method is required to return the loss averaged loss score for the current batch. The test- and validation methods are not required to return anything. Instead, they can use the built-in logging capabilities of the `LightningModule`. Similar to the `.train_step`-method, they should return their scores averaged over the complete batch too.

Model Hyperparameters and checkpoints

Much effort has been put into organizing the hyperparameters of an experiment. Like the train and test routines, PyTorch Lightning binds all hyperparameters that control the model directly to the object. This strategy ensures that saved models also contain the combination of parameters used for training. There are two ways to define the hyperparameters of a model. By default, each argument in the signature of the model's constructor is regarded as a hyperparameter. By calling a `.save_hyperparameters`-method in the constructor, these arguments get serialized into a `.hparams`-attribute. The `.hparams`-attribute is saved as a `YAML`-file for each checkpoint of the model, making it easy to see which parameters were used without loading the whole model. If the constructor contains non-hyperparameters arguments, these can be excluded from serialization using the saving method's `ignore` flag.

Another more explicit way of defining the parameters of a model is to store them all in a dictionary into the constructor and to pass this dictionary to the `-save_hyperparameters`-method. This strategy is suitable in cases where many arguments of the constructor are non-hyperparameters.

Logging

Logging in PyTorch-Lightning is a two-stage procedure. Inside the `LightningModule`, various metrics can be logged at different steps while training using the `.log`- or `log_dict`-methods. The `.log`-methods can log a single score, while the `log-dict`-method can log multiple scores stored in a dictionary (with names as keys and the score as values). These logs are extracted by `Trainer` and written out in various formats (see Trainer section for further details.) One benefit of using an autonomous logging function is that it gives flexibility to the user in deciding when to log which metrics, for example, making it possible to log something only when a condition applies.

6.1.2 `LighningDataModule`

`PyTorchLightning` also comes with a custom solution to bundle data-related operations into a single object. It is called `LighningDataModule` and should contain the code to load and prepare the data for training and testing. A class derived from `LighntningDataModule` must implement four required methods. The `.prepare_data`-method should implement all steps required to load the data and convert it into a correct representation for the model. To return the splits for training, testing and evaluation, there are `.train|.test|.val_loader`-methods. Each of them has to return a `DataLoader` object. Like the `LightningModule`, its data counterpart has the advantage of holding all code to load and prepare the data, which alleviates distribution and publication. Another key feature of the `LightningDataModule` is its ability to adapt to distributed environments. While the `.prepare_data`-method is called once at the beginning of the training, there are also additional `.setup`- and `teardown`-methods. These methods can define operations pre- or post-training data-preparation steps that must be performed independently on each accelerator.

6.1.3 `Trainer`

The `Trainer` object handles the actual training. It receives the model and data (wrapped in Lightning modules) alongside all training-specific hyperparameters, like the number of epochs, the devices to train on, or a list of loggers to log the progress.

The `Trainer` exposes four high-level methods to the user. Each of them triggers either the training, the validation, the prediction of unseen instances, and the hyperparameter-tuning. Like the `LightningModule`, an instance of the `Trainer` is initialized with all hyperparameters relevant to the training, like the batch size or a number of epochs.

The different stages of the training (training and testing)

Extending the `Trainer`

In contrast to the `LightningModule` and `LightningDataModule` the `Trainer` itself is not intended to be customized in any way. Since their respective objects contain all model or data-related code, the `Trainer` is better kept untouched. Instead, if necessary, the functions of the `Trainer` can be extended with callbacks and plugins. Both of them can add custom operations to different stages of the training. Callbacks implement steps that are not strictly necessary for training. Instead, they can be used to define things like logging or applying non-essential operations to the model (i.e., weigh pruning after each epoch) that add new functions but are not required to perform training. On the other hand, plugins are meant to extend the `Trainer` with new functionalities like adding support for new accelerators or computational backends. So by their scope, they are meant to be used by experienced users who need to extend the Trainer. However, since their API is still in beta and subject to changes in the future, it should be used with caution.

Also, it contains a handful of tweaks to improve the results, like gradient accumulation or gradient . In addition to that, the `Trainer` also supports tuning the learning rate and batch size out of the box. Both tuning features must be enabled while initializing the `Trainer` and can be invoked by calling the `.tune`-method.

Logging

While the model defines what measures are logged, the `Trainer` is responsible for writing out these logs. By default, it logs the standard output. In addition to that, it can be extended with additional loggers. PyTorch Lightning provides built-in loggers that log the progress to Tensorboard or other services like Weights and Biases. Further loggers, can be implemented using the Logger base-class. Multiple loggers are passed to the `Trainer` during initialization as a list.

6.1.4 CLI Interface

PyTorch Lightning supports the creation of command-line interfaces through the `LightninArgumentParser` class. This class is an extended version of the parser from the `jsonargparse` module, and it can parse the arguments of Lightning classes and other classes out-of-the-box. This feature enables adding parameters of different modules to the parser effortlessly.

If more flexibility is needed, for example, when only some parameters of an object should be added to the parser, the best practice is to add a method to the object, which adds these arguments to the parser.

```
parser.add_lightning_class_args(ModelCheckpoint, "checkpoint")
parser.add_class_arguments(TensorBoardLogger, nested_key="tensorboard")
parser.add_lightning_class_args(Trainer, "trainer")
parser = PlLanguageModelForSequenceOrdering.add_model_specific_args(parser)
```

6.2 Implementation

6.2.1 Model

Since we do not build our model from scratch, we need to load the pretrained transformer in the LightningModules's constructor. To be able to load different models, we introduce the name of the model as hyperparameters. Since the model is pretrained, we only have to specify two other hyperparameters, namely the learning rate and the id of the target token. Because Huggingface models are also subclasses of the `nn.Module` class, loading the transformer model works flawlessly, and the language model is recognized as a submodule of the `PlLanguageModelForSequenceOrdering` class.

```
class PlLanguageModelForSequenceOrdering(LightningModule):
    def __init__(self, hparams):
        super().__init__()
        self.save_hyperparameters(hparams)
        self.base_model = AutoModelForTokenClassification.from_pretrained(
            self.hparams["model_name_or_path"],
            return_dict=True,
            output_hidden_states=True,
            num_labels=1,
        )
```

Next, we define a single forward step. Again, the logic is pretty simple since we only need to exclude the labels from the inputs for the language model and pass the rest of the input data to the language model to obtain the outputs.

```
    def forward(self, inputs: Dict[Any, Any]) -> Dict[Any, Any]:
        # We do not want to compute token classification loss, so we remove the␣
↪labels temporarily
        labels = inputs.pop("labels")
        outputs = self.base_model(**inputs)

        # And reattach them later on ...
        inputs["labels"] = labels
        return outputs
```

Because we want to compute the loss while training and validating the model, we factor out the loss function into a separate method. Implementation-wise, the loss function is only slightly variated from the original implementation. The only changes are that we retrieve the target token id from the hyperparameters of the model. Also, we draw inspiration from the `transformers` API and add a custom version of the forward method. This method computes both the forward step and the loss. The loss is then attached to the output of the model.

```
    def _compute_loss(self, batch_labels, batch_logits, batch_input_ids) -> float:
        # Since we have varying number of labels per instance,
        # we need to compute the loss manually for each one.
        loss_fn = nn.MSELoss(reduction="sum")
        batch_loss = torch.tensor(0.0, dtype=torch.float64, requires_grad=True)
        for labels, logits, input_ids in zip(
            batch_labels, batch_logits, batch_input_ids
        ):

            # Firstly, we need to convert the sentence indices to regression targets.
            # To avoid exploding gradients, we norm them to be in range 0 <-> 1.
            # labels = labels / labels.max()
            # Also we need to remove the padding entries (-100).
            true_labels = labels[labels != -100].reshape(-1)
            targets = true_labels.float()

            # Secondly, we need to get the logits
            # from each target token in the input sequence
            target_logits = logits[
                input_ids == self.hparams["target_token_id"]
            ].reshape(-1)

            # Sometimes we will have less target_logits
            # than targets due to truncation of the input.
            # In this case, we just consider as many targets as we have logit.
            if target_logits.size(0) < targets.size(0):
```

(continues on next page)

(continued from previous page)

```
                targets = targets[: target_logits.size(0)]

            # Finally we compute the loss for the current instance
            # and add it to the batch loss.
            batch_loss = batch_loss + loss_fn(targets, target_logits)

        # The final loss is obtained by averaging
        # over the number of instances per batch.
        loss = batch_loss / batch_logits.size(0)

        return loss

    def _forward_with_loss(self, inputs):
        outputs = self(inputs)

        # Get sentence indices
        batch_labels = inputs["labels"]
        # Get logits from model
        batch_logits = outputs["logits"]
        # Get logits for all cls tokens
        batch_input_ids = inputs["input_ids"]

        loss = self._compute_loss(
            batch_labels=batch_labels,
            batch_logits=batch_logits,
            batch_input_ids=batch_input_ids,
        )
        outputs["loss"] = loss

        return outputs
```

Using the `_foward_with_loss`-method implementing the `training_step`-method becomes relatively simple. The only thing left to do inside this method is to log the training loss in order to be able to monitor the progress during training.

```
    def training_step(self, inputs: Dict[Any, Any], batch_idx: int) -> float:
        outputs = self._forward_with_loss(inputs)
        loss = outputs["loss"]
        self.log("loss", loss, logger=True)
        return loss
```

Like the `_compute_loss`-method, we only need to slightly adapt the validation metrics' computation to use the model's hyperparameters. Since we want to compute the identical scores for testing and validation, we can also use the `validation_step`-method for testing.

```
    def validation_step(self, inputs, batch_idx):
        outputs = self._forward_with_loss(inputs)

        # Detach all torch.tensors and convert them to np.arrays.
        for key, value in outputs.items():
            if isinstance(value, torch.Tensor):
                outputs[key] = value.detach().cpu().numpy()
        for key, value in inputs.items():
            if isinstance(value, torch.Tensor):
                inputs[key] = value.detach().cpu().numpy()
```

(continues on next page)

(continued from previous page)

```
        # Get sentence indices
        batch_labels = inputs["labels"]
        # Get logits from model
        batch_logits = outputs["logits"]
        # Get logits for all cls tokens
        batch_input_ids = inputs["input_ids"]

        metrics = defaultdict(list)
        for sent_idx, input_ids, logits in zip(
            batch_labels, batch_input_ids, batch_logits
        ):
            sent_idx = sent_idx.reshape(-1)
            input_ids = input_ids.reshape(-1)
            logits = logits.reshape(-1)

            sent_idx = sent_idx[sent_idx != 100]
            target_logits = logits[input_ids == self.hparams["target_token_id"]]
            if sent_idx.shape[0] > target_logits.shape[0]:
                sent_idx = sent_idx[: target_logits.shape[0]]

            # Calling argsort twice on the logits
            # gives us their ranking in ascending order.
            predicted_idx = np.argsort(np.argsort(target_logits))
            tau, pvalue = kendalltau(sent_idx, predicted_idx)
            acc = accuracy_score(sent_idx, predicted_idx)
            metrics["kendalls_tau"].append(tau)
            metrics["acc"].append(acc)
            metrics["mean_logits"].append(logits.mean().item())
            metrics["std_logits"].append(logits.std().item())

        metrics["loss"] = outputs["loss"].item()

        # Add val prefix to each metric name and compute mean over the batch.
        metrics = {
            f"val_{metric}": np.mean(scores).item()
            for metric, scores in metrics.items()
        }
        self.log_dict(metrics, prog_bar=True, logger=True, on_epoch=True, on_
↪step=True)
        return metrics

    def test_step(self, inputs, batch_idx):
        return self.validation_step(inputs, batch_idx)
```

Lastly, we need to implement the `configure_optimizers`-method and add the model's hyperparameter to the parser via the `add_model_specific_args`-method.

```
    def configure_optimizers(self):
        return torch.optim.Adam(params=self.parameters(), lr=self.hparams["lr"])

    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = parent_parser.add_argument_group(
            "PlLanguageModelForSequenceOrdering"
            )
```

(continues on next page)

(continued from previous page)

```
        parser.add_argument(
            "--model.model_name_or_path", type=str, default="bert-base-cased"
        )
        parser.add_argument("--model.lr", type=float, default=3e-5)
        parser.add_argument("--model.target_token_id", type=int, default=101)
        return parent_parser
```

6.2.2 Data

In contrast to the model class, we design our version of the `LightningDataModule` to work with any Huggingface `Dataset`. Most of the work is done by the `.prepare_data`-method, which implements the processing pipeline for the contained dataset Firstly, it applies all functions to prepare the data via the `.map`-method of the `Dataset` class. Afterward, the text data will be tokenized using the passed instance of the tokenizer. Lastly, it is ensured that the dataset's column containing the target is named `labels` to be compliant with standard `transformers` models. Additionally, we implement a method to use the map functionalities of the contained dataset directly. This method allows the manipulation of the data manually since the `.prepare_data`-method is automatically executed by the `Trainer`. The datasets wrapped in this class should already contain train-/ test- and validation-splits. To create batches of the data, we use the default collation function of the transformers library but allow passing a custom collation function.

```
class HuggingfaceDatasetWrapper(LightningDataModule):
    def __init__(
        self,
        dataset: Dataset,
        text_column: str,
        target_column: str,
        tokenizer: PreTrainedTokenizerBase,
        train_batch_size: int = 8,
        eval_batch_size: int = 16,
        mapping_funcs: List[Callable] = None,
        collate_fn: Callable = default_data_collator,
        train_split_name: str = "train",
        eval_split_name: str = "val",
        test_split_name: str = "test",
    ):
        super().__init__()
        self.dataset = dataset
        self.text_column = text_column
        self.target_column = target_column
        self.tokenizer = tokenizer
        self.train_batch_size = train_batch_size
        self.eval_batch_size = eval_batch_size
        self.mapping_funcs = mapping_funcs
        self.collate_fn = collate_fn
        self.train_split_name = train_split_name
        self.eval_split_name = eval_split_name
        self.test_split_name = test_split_name

    def prepare_data(self, tokenizer_kwargs: Dict[str, str] = None):
        # 1. Apply user defined preparation functions
        if self.mapping_funcs:
            for mapping_func in self.mapping_funcs:
                dataset = dataset.map(mapping_func, batched=True)
```

(continues on next page)

(continued from previous page)

```
        # 2. Tokenize the text
        if tokenizer_kwargs is None:
            tokenizer_kwargs = {
                "truncation": True,
                "padding": "max_length",
                "add_special_tokens": False,
            }
        self.dataset = self.dataset.map(
            lambda e: self.tokenizer(e[self.text_column], **tokenizer_kwargs),
            batched=True,
        )
        # 3. Set format of important columns to torch
        self.dataset.set_format(
            "torch", columns=["input_ids", "attention_mask", self.target_column]
        )
        # 4. If the target columns is not named 'labels' rename it
        try:
            self.dataset = self.dataset.rename_column(self.target_column, "labels")
        except ValueError:
            # target column should already have correct name
            pass

    def train_dataloader(self):
        return DataLoader(
            self.dataset[self.train_split_name],
            batch_size=self.train_batch_size,
            collate_fn=self.collate_fn,
        )

    def val_dataloader(self):
        return DataLoader(
            self.dataset[self.eval_split_name],
            batch_size=self.eval_batch_size,
            collate_fn=self.collate_fn,
        )

    def test_dataloader(self):
        return DataLoader(
            self.dataset[self.test_split_name],
            batch_size=self.eval_batch_size,
            collate_fn=self.collate_fn,
        )

    def map(self, *args, **kwargs):
        self.dataset = self.dataset.map(*args, **kwargs)
        return self
```

6.2.3 Complete code

Once again, after factoring out the custom modules, the actual experiment can be implemented in relatively few lines of code. To control the experiment via the command line, we use the `LightningArgumentParser`. We initialize the parser with all arguments from the `Trainer`, `PlLanguageModelForSequenceOrdering`, and `HuggingfaceDatasetWrapper`. Additionally, we add more parameters to give each run a name and control the batch sizes for training and testing. Similar to implementing the experiment with the Huggingface `Trainer`, we need to ensure that the sentences contain the correct special tokens. Replacing these tokens if necessary can be done using the `.map`-method of the `HuggingfaceDatasetWrapper`

```
import json
from os.path import basename
from datasets import load_from_disk
from pytorch_lightning import Trainer, seed_everything
from pytorch_lightning.loggers.tensorboard import TensorBoardLogger
from pytorch_lightning.callbacks import ModelCheckpoint
from pytorch_lightning.utilities.cli import LightningArgumentParser
from transformers import AutoTokenizer

from pl_modules import (
    HuggingfaceDatasetWrapper,
    PlLanguageModelForSequenceOrdering,
    so_data_collator,
)


def main(model_args, trainer_args, checkpoint_args, tensorboard_args, run_args):

    seed_everything(run_args["seed"])

    print("Loading tokenizer.")
    tokenizer = AutoTokenizer.from_pretrained(model_args["model_name_or_path"])

    print("Loading datasets.")
    data = load_from_disk("../data/rocstories")

    # Downsampling for debugging...
    # data = data.filter(lambda _, index: index < 10000, with_indices=True)

    dataset = HuggingfaceDatasetWrapper(
        data,
        text_column="text",
        target_column="so_targets",
        tokenizer=tokenizer,
        mapping_funcs=[],
        collate_fn=so_data_collator,
        train_batch_size=run_args["train_batch_size"],
        eval_batch_size=run_args["val_batch_size"],
    )

    if tokenizer.cls_token != "[CLS]":
        print(
            f"Model does not a have a [CLS] token. Updating the data with token
↪{tokenizer.cls_token} ..."
        )

        def replace_cls_token(entry):
```

(continues on next page)

(continued from previous page)

```
            texts = entry["text"]
            replaced_texts = []
            for text in texts:
                replaced_texts.append(text.replace("[CLS]", tokenizer.cls_token))
            entry["text"] = replaced_texts
            return entry

        dataset = dataset.map(replace_cls_token, batched=True)
        model_args["target_token_id"] = tokenizer.cls_token_id

    print("Loading model.")
    model = PlLanguageModelForSequenceOrdering(hparams=model_args)

    print("Initializing trainer.")
    # Init logger
    tensorboard_logger = TensorBoardLogger(**tensorboard_args)

    # Init callbacks
    callbacks = []
    checkpoint_callback = ModelCheckpoint(**checkpoint_args)
    callbacks.append(checkpoint_callback)

    # Remove default args
    trainer_args.pop("logger")
    trainer_args.pop("callbacks")
    trainer = Trainer(logger=tensorboard_logger, callbacks=callbacks, **trainer_args)

    print("Start training.")
    trainer.fit(model=model, datamodule=dataset)

    print("Start testing.")
    test_results = trainer.test(model=model, datamodule=dataset, ckpt_path=None)
    with open(f"test_results_{model_args['model_name_or_path']}.json", "w") as f:
        json.dump(test_results, f)


if __name__ == "__main__":
    parser = LightningArgumentParser()
    group = parser.add_argument_group()
    group.add_argument("--run.run_name", type=str, default=basename(__file__))
    group.add_argument("--run.seed", type=int, default=0)
    group.add_argument("--run.train_batch_size", type=int, default=8)
    group.add_argument("--run.val_batch_size", type=int, default=16)

    parser.add_lightning_class_args(ModelCheckpoint, "checkpoint")
    parser.add_class_arguments(TensorBoardLogger, nested_key="tensorboard")
    parser.add_lightning_class_args(Trainer, "trainer")
    parser = PlLanguageModelForSequenceOrdering.add_model_specific_args(parser)

    args = parser.parse_args()

    model_args = args.get("model", {})
    trainer_args = args.get("trainer", {})
    checkpoint_args = args.get("checkpoint", {})
    tensorboard_args = args.get("tensorboard", {})
    run_args = args.get("run", {})
```

(continues on next page)

(continued from previous page)

```
    main(model_args, trainer_args, checkpoint_args, tensorboard_args, run_args)
```

6.3 Conclusion

Pytorch Lightning's goal is not to hide complexity from the user. Instead, it provides an API that helps to structure the complexity into a sequence of single steps. This approach is constructive when designing custom models from scratch or implementing new training regimes that differ from the standard training loop. This flexibility comes at the cost of friendliness to beginners. People who have little experience with PyTorch itself will quickly be overwhelmed by PyTorch Lightning API with vast possibilities to customize steps manually. Even though the documentation is extensive and covers nearly all aspects of the library in great detail, it can be frustrating sometimes that there are multiple ways to achieve the same behavior, and there is little to no guidance in choosing between the different parts. Like most modern deep learning frameworks, PyTorch Lightning is rapidly evolving, and thus many parts of it are either in beta and subject to significant changes in the future or deprecated. Unfortunately, this is also noticeable when searching the web for further advice since many tips or tutorials quickly become outdated. Nevertheless, despite these limitations for beginners, experienced users can benefit from using PyTorch Lightning. Not only because of the additional features like built-in logging, tuning, or other tweaks but mainly because the well-thought API enforces them to write self-contained models that contain all the logic for experimenting with them. This approach effortlessly enables sharing of models and also alleviates maintainability.

CHAPTER

SEVEN

POUTYNE

Compared to the other two frameworks, Poutyne (Paradis *et al.* [2020]) has a more narrow scope. Instead of trying to make the training of a fixed set of models as easy as possible like Huggingface `Trainer`, or facilitating the creation and training of custom models like PyTorch Lightning, it tries to bring the ease of the Keras API from the realms of Tensorflow to the world of PyTorch. The benefits of the Keras API are its simplicity and orientation at well-established machine learning frameworks like Scikit-Learn. This simplicity lowers the barrier of entry for beginners because it lowers the amount of time needed to get hands-on training for their first model. The following exemplary listing shows the typical workflow in Poutyne.

```
from poutyne import Model

...

network = make_network()
X_train, y_train = load_data(subset="train")
X_val, y_val = load_data(subset="validation")
X_test, y_test = load_data(subset="test")

model = Model(
    network,
    "sgd",
    "cross_entropy",
    batch_metrics=["accuracy"],
    epoch_metrics=["f1"],
    device="cuda:0"
)

model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=5,
    batch_size=64
)

results = model.evaluate(X_test, y_test, batch_size=128)
```

Like Keras, Poutyne automates many steps for standard cases like the optimizer configuration or the loss function. However, Poutyne does not mimic the whole Keras API but only the training part. The model's creation still has to be done in plain PyTorch, which is generally a bit trickier than Keras because the dimensions of all layers have to be chosen manually. In addition to the training functions, Poutyne also provides utilities to conduct and save whole experiments and utilities for creating checkpoints, logging, scheduling of the learning rate, and multi-device training.

7.1 Classes

7.1.1 Model

The `Model` class wraps a neural network alongside an optimizer, loss function, and validation metrics. It exposes `.fit`-, `.evaluate`-, and `.predict`-methods for training, evaluation, and inference. Each of these methods exits in different variations that consume the data either as a list of batches, PyTorch `Dataset`, or as a generator yielding batches. Additional hyperparameters, like the batch size, or the number of epochs to train, can be passed the methods directly.

7.1.2 Experiment

The `Experiment` class is an extended version of the `Model` class that comes with helpful additions for conducting deep learning experiments. Like the `Model` class, an `Experiment` is equipped with the neural network, optimizer, loss function, and metrics into a single object and has methods to start the training, evaluation, or prediction. In contrast to the `Model` class, which only intends to do basic training, the `Experiment` class provides additional features to organize and track the progress. For example, it supports logging the progress to various formats, like a CSV table or Tensorboard. Monitoring allows the `Experiment` class to save checkpoints of the model that perform best concerning one of the validation metrics. Also, it saves all the intermediate results and tracked values to the disk by default. The `Experiment` class can automatically configure all metrics and the loss function for the two primary task types, classification, and regression.

7.1.3 Data

Poutyne is data agnostic meaning, that it does not provide any tooling to load, process, and store the training data. The only requirements are that the data comes in one of the supported formats and that each batch consists of two objects: one that holds the training data and one that contains the label.

7.2 Additional Features

7.2.1 Metrics

Poutyne has a custom API for implementing metrics. It distinguishes between two types of metrics, batch metric, and epoch metrics. Batch metrics are computed per batch and averaged to obtain the results for one single epoch. Epoch metrics are computed on the gathered results of one entire epoch. Thus, they are a good choice for measures that would suffer from averaging over the batch results, like the F-score. Poutyne provides predefined metrics for both types. But, unfortunately, they only cover classification tasks. There are two options to add other metrics. Either they have to be implemented manually or taken from Scikit-Learn and made compatible using a built-in wrapper class. Metrics are passed to `Model` or `Experiment` during their initialization.

7.2.2 Callbacks

Callbacks are intended to extend the functions of the `Model` or `Experiment` class. Like the callbacks from the other frameworks, they have access to the model's current state and can perform actions at various steps while training. There are many predefined callbacks available that perform all kinds of tasks, ranging from logging, keeping track of gradients, scheduling the learning, creating checkpoints, to sending notifications to inform clients about the progress of the training.

7.3 Implementation

Even though the `transformers` models are incompatible with vanilla Poutyne, integrating it does not require complicated changes. Most of the required adaptions change the data in order to convert betweeN the dictionary-based data model of the `transformers` library and Poutyne's more classical `X, y` format for input data and targets. Since these changes are task agnostic, we factored most of these adaption tools out of the project into a small standalone library.[1]

7.3.1 Data

We create a custom data collator to convert the data for an experiment from the Hugginface `Dataset` format into a Poutyne compliant representation. The main task of the collator is to convert each batch of dictionaries into batches containing tuples of training data and targets. To do so, the `TransformersCollator` copies one or multiple entries from the input dictionaries into the target objects. Depending on the number of keys, this object is either a single tensor or a dictionary. Additionally, with the `remove_labels`-parameter, the fields that get copied to the target object can be removed from the model's input. By default, they are retained in the input data. This functionality enables using the internal computation of the loss of standard models while also being able to use the built-in metrics of Poutyne for monitoring the training. Other collation operations are handled by the default collator from `transformers` or by a custom function.

```
from typing import Any, Callable, Dict, List, Tuple, Union

import torch
from transformers import default_data_collator


class TransformerCollator:
    def __init__(
        self,
        y_keys: Union[str, List[str]] = None,
        custom_collator: Callable = None,
        remove_labels: bool = False,
    ):
        self.y_keys = y_keys
        self.custom_collator = (
            custom_collator if custom_collator is not None else default_data_collator
        )
        self.remove_labels = remove_labels

    def __call__(self, inputs: Tuple[Dict]) -> Tuple[Dict, Any]:
        batch_size = len(inputs)
        batch = self.custom_collator(inputs)
        if self.y_keys is None:
            y = torch.tensor(float("nan")).repeat(batch_size)
        elif isinstance(self.y_keys, list):
```

(continues on next page)

[1] poutyne-transformers

(continued from previous page)

```
            y = {
                key: batch.pop(key)
                if "labels" in key and self.remove_labels
                else batch.get(key)
                for key in self.y_keys
            }
        else:
            y = batch.pop(self.y_keys) if self.remove_labels else batch.get(self.y_
↪keys)
        return batch, y
```

7.3.2 Model

As stated in the Prerequisites chapter, tokenizers return a dictionary of data that contains all data required to be fed into the language model unpacked as keyword arguments.

```
from transformers import AutoModel, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
model = AutoModel.from_pretrained("bert-base-cased")

inputs = tokenizer("Poutyne is inspired by Keras", return_tensors="pt")
print(model(**inputs).keys())
```

```
odict_keys(['last_hidden_state', 'pooler_output'])
```

Poutyne instead passes the data to the model in the same format it receives it. To make sure that the data is unpacked correctly, we create a wrapper class. It is also a subclass of the `nn.Module` to ensure that all parameters of the capsulated model can be accessed. Apart from the data handling, this class also exposes the custom `save_pretrained`-model of the underlying `transformers` model. This way, it is possible to create checkpoints of the trained model that can be loaded and used in the `transformers` ecosystem.

```
from typing import Any, Dict
from torch import nn
from transformers import PreTrainedModel


class ModelWrapper(nn.Module):
    def __init__(self, transformer: PreTrainedModel):
        super().__init__()
        self.transformer = transformer

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}({repr(self.transformer)})"

    def forward(self, inputs) -> Dict[str, Any]:
        return self.transformer(**inputs)

    def save_pretrained(self, *args, **kwargs) -> None:
        self.transformer.save_pretrained(*args, **kwargs)
```

7.3.3 Loss

In Poutyne the loss function receives the output of the model and the targets. When using default models, neither of both has to be used to obtain the loss, since we can extract the internal loss from the model's output. In our case we have to implement a function that computes the loss on our own. Since we do not have access to the model or the tokenizer, we have to create a loss function that stores the id of the current target token. For that, we opt for creating a class that holds this id as an attribute and computes the loss via its `__call__` method.

```
class PoutyneSequenceOrderingLoss:
    def __init__(self, target_token_id):
        self.target_token_id = target_token_id

    def __call__(self, outputs, targets) -> float:
        batch_labels = targets["labels"]
        batch_logits = outputs["logits"]
        batch_input_ids = targets["input_ids"]

        # Since we have varying number of labels per instance, we need to compute the␣
↪loss manually for each one.
        loss_fn = nn.MSELoss(reduction="sum")
        batch_loss = torch.tensor(0.0, dtype=torch.float64, requires_grad=True)
        for labels, logits, input_ids in zip(
            batch_labels, batch_logits, batch_input_ids
        ):
            # Firstly, we need to convert the sentence indices to regression targets.
            # To avoid exploding gradients, we norm them to be in range 0 <-> 1
            # Also we need to remove the padding entries (-100)
            true_labels = labels[labels != -100].reshape(-1)
            targets = true_labels.float()

            # Secondly, we need to get the logits from each target token in the input␣
↪sequence
            target_logits = logits[input_ids == self.target_token_id].reshape(-1)

            # Sometimes we will have less target_logits than targets due to trunction␣
↪of the input
            # In this case, we just consider as many targets as we have logits
            if target_logits.size(0) < targets.size(0):
                targets = targets[: target_logits.size(0)]

            # Finally we compute the loss for the current instance and add it to the␣
↪batch loss
            batch_loss = batch_loss + loss_fn(targets, target_logits)

        # The final loss is obtained by averaging over the number of instances per␣
↪batch
        loss = batch_loss / batch_logits.size(0)

        return loss
```

7.3.4 Metrics

Unlike the Huggingface `Trainer`, which expects all external metrics a single function to compute them all at once, in Poutyne, the `Model` or `Experiment` classes are equipped with multiple single functions for each metric. Like the loss, functions that compute other performance metrics receive the model's output alongside the targets (extracted by collation function). Because `transformer` models return not only the logits or predictions of a model but also other things, it is not possible to use Poutynes built-in metrics out of the box. They expect the output to be a single tensor containing the logits of the model, so we create a wrapper for metric functions that extracts them from the output and passes them to the metric.

```
from typing import Any, Callable, Dict

class MetricWrapper:
    def __init__(self, metric: Callable, pred_key: str = "logits", y_key: str = None):
        self.metric = metric
        self.pred_key = pred_key
        self.y_key = y_key
        self._set_metric_name(metric)

    def _set_metric_name(self, metric):
        self.__name__ = metric.__name__

    def __call__(self, outputs: Dict[str, Any], y_true: Any):
        y_pred = outputs[self.pred_key]
        if self.y_key is not None:
            y_true = outputs[self.y_key]
        return self.metric(y_pred, y_true)
```

Since the logging components of Poutyne infer the name of the metric by assessing the class name of their functions, we need to set the `__name__`-attribute of our wrapper instance with the name of the contained metric.

To implement our sentence ordering metrics, we adopt our existing code to return a function for each metric.

```
import numpy as np
from collections import defaultdict
from functools import partial
from sklearn.metrics import accuracy_score
from scipy.stats import kendalltau

def make_compute_metrics_functions(target_token_id) -> Callable:
    def compute_ranking_func(
        outputs: Dict, targets: Any, metric_key: str
    ) -> Dict[str, float]:
        batch_sent_idx = targets["labels"].detach().cpu().numpy()
        batch_input_ids = targets["input_ids"].detach().cpu().numpy()
        batch_logits = outputs.detach().cpu().numpy()

        metrics = defaultdict(list)
        for sent_idx, input_ids, logits in zip(
            batch_sent_idx, batch_input_ids, batch_logits
        ):
            sent_idx = sent_idx.reshape(-1)
            input_ids = input_ids.reshape(-1)
            logits = logits.reshape(-1)

            sent_idx = sent_idx[sent_idx != 100]
            target_logits = logits[input_ids == target_token_id]
```

(continues on next page)

(continued from previous page)

```
            if sent_idx.shape[0] > target_logits.shape[0]:
                sent_idx = sent_idx[: target_logits.shape[0]]
            # Calling argsort twice on the logits gives us their ranking in ascending␣
↪order
            predicted_idx = np.argsort(np.argsort(target_logits))
            tau, pvalue = kendalltau(sent_idx, predicted_idx)
            acc = accuracy_score(sent_idx, predicted_idx)
            metrics["kendalls_tau"].append(tau)
            metrics["acc"].append(acc)
            metrics["mean_logits"].append(logits.mean())
            metrics["std_logits"].append(logits.std())
        metrics = {metric: np.mean(scores) for metric, scores in metrics.items()}
        return metrics[metric_key]

    metrics = []
    for metric in ("acc", "kendalls_tau", "mean_logits", "std_logits"):
        metric_func = partial(compute_ranking_func, metric_key=metric)
        metric_func.__name__ = metric
        metrics.append(metric_func)
    return metrics

metrics = [
        MetricWrapper(func)
        for func in make_compute_metrics_functions(0)
    ]
print([metric.__name__ for metric in metrics])
```

```
['acc', 'kendalls_tau', 'mean_logits', 'std_logits']
```

Additionally, we add two functions to track the mean and standard deviation of the logits to monitor whether the regression can fit the desired indices or only learns their average, which lies around `2.5`.

7.3.5 Complete code

Once again, we factor out our adaptions into an external module and implement the rest of the experiment. Due to Poutyne's lack of tooling for creating a command-line interface, this experiment is only configurable via hard-coding the parameters into the source. The rest of the code is mainly similar to the other two frameworks.

```
import json
from poutyne.framework import experiment
from torch.optim import AdamW
from poutyne import (
    set_seeds,
    TensorBoardLogger,
    TensorBoardGradientTracker,
    Experiment,
)
from poutyne_transformers import ModelWrapper, MetricWrapper, TransformerCollator
from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import DataLoader
from transformers import AutoTokenizer, AutoModelForTokenClassification
from datasets import load_from_disk
from poutyne_modules import (
```

(continues on next page)

(continued from previous page)

```
    make_tokenization_func,
    PoutyneSequenceOrderingLoss,
    make_compute_metrics_functions,
    so_data_collator,
)


if __name__ == "__main__":
    set_seeds(42)

    MODEL_NAME_OR_PATH = "bert-base-cased"
    LEARNING_RATE = 3e-5
    TRAIN_BATCH_SIZE = 8
    VAL_BATCH_SIZE = 16
    DEVICE = 0
    N_EPOCHS = 3
    SAVE_DIR = "experiments/rocstories/bert"

    print("Loading model & tokenizer.")
    tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME_OR_PATH)
    transformer = AutoModelForTokenClassification.from_pretrained(
        MODEL_NAME_OR_PATH, return_dict=True, num_labels=1
    )

    print("Loading & preparing data.")
    dataset = load_from_disk("../data/rocstories/")

    if tokenizer.cls_token != "[CLS]":
        print(
            f"Model does not a have a [CLS] token. Updating the data with token
 ↪{tokenizer.cls_token} ..."
        )

        def replace_cls_token(entry):
            texts = entry["text"]
            replaced_texts = []
            for text in texts:
                replaced_texts.append(text.replace("[CLS]", tokenizer.cls_token))
            entry["text"] = replaced_texts
            return entry

        dataset = dataset.map(replace_cls_token, batched=True)

    tokenization_func = make_tokenization_func(
        tokenizer=tokenizer,
        text_column="text",
        add_special_tokens=False,
        padding="max_length",
        truncation=True,
    )
    dataset = dataset.map(tokenization_func, batched=True)

    dataset = dataset.rename_column("so_targets", "labels")

    dataset = dataset.remove_columns(
        ["text", "storyid", "storytitle"] + [f"sentence{i}" for i in range(1, 6)]
```

(continues on next page)

(continued from previous page)

```
    )
    dataset.set_format("torch")

    collate_fn = TransformerCollator(
        y_keys=["labels", "input_ids"],
        custom_collator=so_data_collator,
        remove_labels=True,
    )

    train_dataloader = DataLoader(
        dataset["train"], batch_size=TRAIN_BATCH_SIZE, collate_fn=collate_fn
    )
    val_dataloader = DataLoader(
        dataset["val"], batch_size=VAL_BATCH_SIZE, collate_fn=collate_fn
    )
    test_dataloader = DataLoader(
        dataset["test"], batch_size=VAL_BATCH_SIZE, collate_fn=collate_fn
    )

    print("Preparing training.")
    wrapped_transformer = ModelWrapper(transformer)
    optimizer = AdamW(wrapped_transformer.parameters(), lr=LEARNING_RATE)
    loss_fn = PoutyneSequenceOrderingLoss(target_token_id=tokenizer.cls_token_id)

    metrics = [
        MetricWrapper(func)
        for func in make_compute_metrics_functions(tokenizer.cls_token_id)
    ]

    writer = SummaryWriter("runs/roberta/1")
    tensorboard_logger = TensorBoardLogger(writer)
    gradient_logger = TensorBoardGradientTracker(writer)

    experiment = Experiment(
        directory=SAVE_DIR,
        network=wrapped_transformer,
        device=DEVICE,
        logging=True,
        optimizer=optimizer,
        loss_function=loss_fn,
        batch_metrics=metrics,
    )

    experiment.train(
        train_generator=train_dataloader,
        valid_generator=val_dataloader,
        epochs=N_EPOCHS,
        save_every_epoch=True,
    )

    test_results = experiment.test(test_generator=test_dataloader)
    with open(f"test_results_{MODEL_NAME_OR_PATH}.json", "w") as f:
        json.dump(test_results, f)
```

7.4 Conclusion

Poutyne provides a well thought and, most of all easy to understand framework to train neural networks. Like its conceptual role model Keras, this simplicity is achieved by strict design decisions, like the `X, y` format for data. While this strictness is helpful for beginners because they only have to learn one way of doing things, it comes at the cost of being hard to adapt to other frameworks or unintended tasks. Luckily, the necessary steps to adapt it to `transformers` and our task are simple and can be reused for most other cases. Since Poutynes mimics the Keras-API, its additional features are much more limited than the other frameworks. Even basic techniques like gradient accumulation are not supported. Depending on the use case, this limited scope might be a deal-breaker for experienced users or complex tasks, but on the other hand, it makes getting started with the framework much more manageable. This accessibility is underlined by the documentation's quality, which covers all aspects of the framework in concise and easily understandable manners without losing itself in the depths of technical details. Yet, there is also potential for further improvements. The lack of support for creating-command line interfaces could force users to migrate to another framework as soon as they need to retrain a model regularly. Currently, the scope of the framework is heavily skewed towards sequence classification tasks. For example, all built-in metrics measure the quality of a classification model. Widening the range of tasks that could be implemented without further extensions would help beginners get into deep learning. A possible improvement that falls more into the category of wishful thinking would be that Poutyne would mimic not only the training parts of the Keras API. If Poutyne would also introduce the ease of building neural networks without manually adjusting each layer's dimensionality, it would significantly contribute to the community.

CHAPTER

EIGHT

EXPERIMENTAL RESULTS

To see if our custom task architecture is able to order the sentences, we run the experiment using the ROCStories Dataset with three different pretrained language models. Also, we run the experiment using the same set of parameters with each framework to check if they perform consistently or if any under-the-hood magic influences the scores.

8.1 Hyperparemters

We employ `distilbert-base-cased`, `bert-base-cased`, and `roberta-base` as pretrained models. Intuitively, we expect `roberta-Base` to perform best, with `bert-case` reaching a tight second place and `distilbert` to fall short behind the large two models. We use the AdamW optimizer with a learning rate of $3e-5$. We finetune for 3 epochs and validate our models on the test set while the validation set is only used for tracking the progress during training. Furthermore, we use the same random seed across all models and frameworks. Due to varying model sizes, we use different batch sizes to fit the model on the GPU. To ensure that the batch sizes do not affect the performance, we use gradient accumulation with the Huggingface `Trainer` and PyTorch Lightning to ensure that each model makes the same number of backward steps. For Poutyne, which does not support gradient accumulation, we chose the largest batch size possible for each model. In addition to these basic parameters, each framework has a set custom parameter that we leave untouched and use the default configuration.

8.2 Results

Table 8.1: Results

Framework	Model	Loss	Accuracy	τ_{Kendall}
Huggingface `Trainer`	`bert-base-cased`	2.447	0.699	0.792
	`roberta-base`	1.619	0.786	0.861
	`distilbert-base-cased`	2.800	0.653	0.753
PyTorch Lightning	`bert-base-cased`	2.621	0.696	0.785
	`roberta-base`	1.755	0.776	0.853
	`distilbert-base-cased`	2.933	0.651	0.748
Poutyne	`bert-base-cased`	2.714	0.687	0.776
	`roberta-base`	2.106	0.748	0.830
	`distilbert-base-cased`	3.024	0.645	0.742

Table 8.1 shows the results of the runs. As expected, `roberta-base` achieved the best results. However, the margin by which it outperforms the standard `bert-base-cased` model is surprising. In contrast to Bert models, Roberta models do not use the next sentence prediction objective during their finetuning stages. Next sentence prediction is the task of deciding if two consecutive sentences in the input sequence are natural successors or not. Intuitively, the knowledge obtained from this task should help order sentences too. Obviously, without additional inspection, each further presumption is mere speculation. Still, it seems like the fact that Roberta models are pretrained on a larger dataset outweighs the missing next sentence objective during pretraining.

Comparing the results by each framework, the Huggingface `Trainer` achieves a clear victory. It reaches the best results across all models and, depending on the metric, outperforms the other frameworks significantly. However, while PyTorch Lightning manages to keep up the the `Trainer`, Poutyne falls clear behind its competitors, especially when training the larger models. This difference in performance indicates the benefit of gradient accumulation when training models that only allow small batch sizes. However, although more subtle, the gap between the Huggingface `Trainer` and PyTorch Lightning is surprising since both frameworks offer roughly the same feature set. Different parameters can probably explain it since we only changed some parameters and used the default values provided by the developers for the rest. However, due to its focus on language models, the `Trainer` should have a preset of parameters well optimized for language models. The most plausible candidate for explaining the observed difference is the learning schedule. While PyTorch Lightning uses a fixed learning rate if not specified otherwise, the Huggingface `Trainer` employs a learning scheme with a linear decay and additional warmup steps. This strategy, which is proven to be helpful when training large models, might cause the superior results of the `Trainer`.

Despite their differences, it has to be stated that all frameworks achieved robust results. The relative order of the different models is consistent across all frameworks. Therefore, the results of each framework are sufficient not to shade the capabilities of the models or language models in general for this task.

Comparing to the other works, which employ a pretrained language model, we beat the regression baseline from Kumar *et al.* [2020] ($\tau_{\text{Kendall}} = 0.736$) by quite a large margin but still fall clearly behind the current state of art model by Zhu *et al.* [2021] ($\tau_{\text{Kendall}} = 0.849$). Since the results of both papers were achieved using a standard Bert model, we also have to compare them against our Bert scores. By looking at the Roberta scores, it becomes clear that choosing a larger or better language model seems to have a large influence regardless of the approach. But all of these results have to be taken with a grain of salt since we only ran the experiment once using a fixed random seed and a custom train-test-validation split, making our results less reliable.

CHAPTER

NINE

CONCLUSION

Due to their different scopes labeling one of the presented frameworks as the "best one" would paint a misleading picture. Obviously, the built-in `Trainer` of the `transformers` library is optimally aligned with the rest of the library, which facilitates the training of language models in many cases. This ease, combined with its optimized set of predefined parameters, makes it the best choice when training standard models. Even in cases like ours, where a standard model is combined with a custom loss, the `Trainer` requires few adaptions. However, the goal to allow those adaptions with as few lines of code as possible also has some drawbacks. Conceptually, the separation of concerns between integral parts of the model and additional logic is less strict. For example, by default, `transformers` models incorporate the loss function into their heads. But if this loss should be discarded, the custom loss function is bound to the custom subclass of the `Trainer`. This scattering leads to an implicit separation of the model and its loss. Without access to the `Trainer` subclass, continuing the model's training is impossible. More gravely, this fact is hidden too, since loading the model looks like a standard token classification model and even returns a loss score when fed with the correct data. Of course, one could argue that it is possible to create proper custom models with an own head, which would be a cleaner way to implement such a model. But since making a custom model is a rather complex process, it would also render the advantage of the `Trainer` irrelevant.

From a conceptual point of view, PyTorch Lightning's approach is far more sustainable since its API forces to structure the code into mostly self-contained modules. While requiring more manual implementation upfront, this approach leads to better code quality and makes models and datasets easily interchangeable and thus more reusable. However, it also expects the user to have a profound knowledge of PyTorch itself, alongside a deep mental model of how a neural network is trained since there are no shortcuts. Users who fit these requirements can benefit from PyTorch Lightning's strict specifications. While these specifications determine the whole process of building neural networks, users do not lose much freedom because the framework is highly flexible and can be modified to a great extend. Yet, there are drawbacks when working with `transformers` and PyTorch Lightning. Most notably, the differences in the serialization of models complicate the process of creating checkpoints that can be used interchangeably between PyTorch Lightning and the Huggingface ecosystem. Also, PyTorch Lightning, like most deep learning frameworks, evolves fast and is updated frequently. This speed can lead to issues that are hard to understand and fix. For example, on the machine used to run the experiments of this work, only one of the six available computational backends that govern Multi-GPU training worked reliably. Using the other ones either led to freezes while training, exceptions that aborted the training complete, or degraded results because the data was corrupted. These problems were exacerbated because PyTorch code is notoriously hard to debug since most of the computations are done outside the Python runtime and thus hard to access with standard debuggers. Nonetheless, all the available extension and hyperparameter tuning functions justify the usage of PyTorch Lightning because once the initial setup is running, improving the results is easy and does not require much work.

To be fair, it has to be stated that including Poutyne in this work is unfair since its scope is much more narrow, and, by its intention, it does not try to offer the same set of functions as both other frameworks. Instead of focusing on its lacks, it becomes clear why Poutyne can be helpful by looking at its conceptual design. Its conceptual paragon Keras is the most beginner-friendly library to get started with deep learning. Since Keras dropped support for other backends like PyTorch, this easy is only available for Tensorflow. However, since most deep learning research is done in PyTorch nowadays, learning Tensorflow has become less attractive because beginners will have to switch from Tensorflow to PyTorch at some point as they progress. So because learning PyTorch is inevitable for most users, it would be beneficial to be able to start right away with it. Without being anywhere near as mature as Keras, Poutyne has the potential of offering a real alternative for it in PyTorch. Additionally, a high-level API that enables the quick training of models effortlessly is

also attractive for experienced users who want to test a prototype. As our results showed, the results won't be as best as possible, but they are sufficient enough to give a first impression. Like Keras, Poutyne is designed to work best with models built from scratch, but it is easy to adapt the framework to work with pretrained models of all sorts. The library `poutyne-transformers` might act as a proof-of-concept for further adjustments.

In conclusion, this work showed that all three of the frameworks have a reason for existence. Choosing between PyTorch Lightning and the Hugginface `Trainer` highly depends on the requirements of the project. PyTorch Lightning is a good choice for models that are intended to go into production. At the same time, the `Trainer` is a great choice when doing research where the speed of iteration outweighs the sustainability of the code. Poutyne is an excellent choice for beginners who want to get into deep learning and start to train models quickly without having to learn complex frameworks.

BIBLIOGRAPHY

[Fal19] WA Falcon. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning Cited by*, 2019.

[KBKR20] Pawan Kumar, Dhanajit Brahma, Harish Karnick, and Piyush Rai. Deep attentive ranking networks for learning to order sentences. *CoRR*, 2020.

[MOBrienR18] David McClure, Shayne O'Brien, and Deb K. Roy. Context is key: new approaches to neural coherence modeling. *ArXiv*, 2018.

[MCH+16] Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. A corpus and evaluation framework for deeper understanding of commonsense stories. 2016. arXiv:1604.01696.

[PBG+20] Frédérik Paradis, David Beauchemin, Mathieu Godbout, Mathieu Alain, Nicolas Garneau, Stefan Otte, Alexis Tremblay, Marc-Antoine Bélanger, and François Laviolette. Poutyne: A Simplified Framework for Deep Learning. 2020. https://poutyne.org.

[WDS+19] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: state-of-the-art natural language processing. *CoRR*, 2019. URL: https://huggingface.co/transformers/index.html.

[YSS+19] Yongjing Yin, Linfeng Song, Jinsong Su, Jiali Zeng, Chulun Zhou, and Jiebo Luo. Graph-based neural sentence ordering. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 5387–5393. International Joint Conferences on Artificial Intelligence Organization, 7 2019. URL: https://doi.org/10.24963/ijcai.2019/748, doi:10.24963/ijcai.2019/748.

[ZNZ+21] Yutao Zhu, Jian-Yun Nie, Kun Zhou, Shengchao Liu, and Pan Du. BERT4SO: neural sentence ordering by fine-tuning BERT. *CoRR*, 2021.