Zelda Rose: a tool for hassle-free training of transformer models

Loïc Grobol
MoDyCo, CNRS, Université Paris Nanterre
Lattice, ENS, CNRS and Université Sorbonne Nouvelle
lgrobol@parisnanterre.fr

Abstract

Zelda Rose is a command line interface for pretraining transformer-based models. Its purpose is to enable an easy start for users interested in training these ubiquitous models, but unable or unwilling to engage with more comprehensive — but more complex — frameworks and the complex interactions between libraries for managing models, datasets and computations. Training a model requires no code on the user's part and produce models directly compatible with the HuggingFace ecosystem, allowing quick and easy distribution and reuse. A particular care is given to lowering the cost of maintainability and future-proofing, by making the code as modular as possible and taking advantage of third-party libraries to limit ad-hoc code to the strict minimum.

1 Introduction

Since their advent in machine translation (Vaswani et al., 2017) and as a mean to obtain contextual word representations (Devlin et al., 2019), transformer models have become ubiquitous in Natural Language Processing. The latter use in particular is almost impossible to avoid to develop state-of-the-art NLP systems, the usual workflow being a self-supervised *pretraining* step using unannotated data, followed by a *fine-tuning* step for a downstream task, either as a module in a larger neural architecture or as an end-to-end predictor.

Using these models and *fine-tuning* them on downstream tasks has been made easier by libraries such as *AllenNLP* (Gardner et al., 2018), *FairSeq* (Ott et al., 2019), *MaChAmp* (van der Goot et al., 2021), *Trankit* (Nguyen et al., 2021)… and most of all *Transformers* (Wolf et al., 2020). However, among these, only *FairSeq* and *Transformers* provide interfaces to *pretrain* them.

While these libraries have considerably lowered the barrier of entry for using these models, producing new ones remains an involved process. In particular training existing models on new data is not trivial, making it hard to develop models for new languages or specialty domains. This difficulty is largely due to the number of moving pieces and options required for training these models with limited resources in a reasonable time.

Zelda Rose is meant to make pretraining transformers models as simple as possible, in particular for users who would benefit from being able to train or refine them on their own specialized domains, but who are not necessarily interested in controlling or customizing every possible aspect of the training process. This include among researchers in domains other than NLP, software engineers in the process of porting existing NLP tools to new languages, and, generally speaking, *consumers* of transformer models as opposed to *researchers* interested in improving the models themselves. It must therefore be a small tool, with a limited and clearly defined purpose, as easy to use and cheap to maintain as possible.

To this end, pretraining a transformer model using Zelda Rose does not require writing any code (although it is easy to write code to extend its capability or customize it), but running a simple command. The configuration is done with entirely with configuration files and command line options. It can use any local datasets, models and configurations as well as refer to HuggingFace Hub repository. It is also transparently compatible with the SLURM scheduler to make it easier to run on computing clusters.

2 Related works

To our knowledge, only two mainstream libraries providing interfaces for pretraining transformer models are *FairSeq* (Ott et al., 2019) and *Transformers* (Wolf et al., 2020) (via its Trainer module). Both of these are complete frameworks, providing model implementations, data process-

Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023), pages 54–58

ing tools and training interfaces. While this allows for a complete customization of the architecture of the models and of the training process, it can makes it less straightforward to train a known model on new data or to get a precise sense of what is happening during training, since it requires an extensive knowledge of their often convoluted code bases. In other words, while they are essential tools to build on, they are not necessarily the most ergonomic for the need of all users. The tight coupling of their model training utilities with their library code also makes it hard to extend them, for instance to make them compatible with new hardware platforms or training techniques.

To avoid these complications, some works (such as He et al. (2020) for DeBERTa), chose to build projects entirely around the purpose of training a single model, writing ad-hoc code more or less from scratch. This obviates the need for learning all the details of a specific framework, but is much more involved in terms of engineering to design the model and training code, run them and ensure the reusability of the resulting artifacts. Overall, this model would not be suitable for the use cases that we target with Zelda Rose.

Between these two extremes of being completely integrated in an end-to-end framework or to build from ad-hoc code, the choice we made for Zelda Rose is a loose coupling with frameworks. In practice, this means that users can train new models by providing a configuration and a dataset to a command line tool, which trains a model using only high-level interfaces of third-party libraries, each specialized in a specific aspect. For instance we use *Pytorch-Lightning* (Falcon and The PyTorch Lightning team, 2019) to manage the training process, while the models implementations come from *Transformers*, which allows to benefits from all their respective innovations while avoiding being restricted by their limitations.

3 Design

3.1 User interface

From the point of view of a user, Zelda Rose mainly consists of a command line interface, which takes parameters related to the *model architecture*, *task and training configurations* and *training platform*. In its most basic form, it trains a model on a masked language task using the same hyperparameters as Liu et al. (2019):

```
zeldarose transformer \
    --tokenizer roberta_base \
    --model-config roberta_base \
    --val-text dev_corpus.txt \
    train_corpus.txt
```

The training parameters can be customized by passing a configuration file in the TOML format. For instance the default configuration would be

```toml
type = "mlm"

[task]
change_ratio = 0.15
mask_ratio = 0.8
switch_ratio = 0.1

[tuning]
batch_size = 64
betas = [0.9, 0.98]
epsilon = 1e-8
learning_rate = 1e-4
lr_decay_steps = 1048567
warmup_steps = 1024
weight_decay = 1e-5
```

The use of configuration files rather than command line flags or environment variables help keeping track of the settings used (they can be directly redistributed with the models) for documentation and reproduction. They are also easier to version and to validate (in our case via *Pydantic* (Colvin et al., 2023)).

On the other hands, parameters related to the training platform are given as command line optiuons, since they are specific to each invocation. For instance `--num-devices` specifies the number of devices (GPU, CPU cores…) used for a training run. Other options include the type of devices to use, the number of nodes to use when running in SLURM…

So far, the tasks implemented are masked language modeling (inspired by BERT Devlin et al. (2019)), replaced token detection (from ELECTRA (Clark et al., 2019) and DeBERTa v3 (He et al., 2021)) and span-masking denoising (from mBART (Liu et al., 2020)). Not all hyperparameters are configurable and some opinionated choices are made (for instance at this point the gradient descent algorithm used is AdamW (Loshchilov and Hutter, 2019)) in order to keep the configuration space manageable, which sim-

plify the choice of a setting for user and reduces the maintainability burden.

The model configurations given in input refer to *Transformers* models, which can be loaded either from a local file or from a repository on HuggingFace Hub[1]. This allow an easy reference to most popular models. Users can also ask for the initialization of their model with already pretrained weights for post-training (Zhuang et al., 2021), which has been shown to significantly improve model performances on domain-specific tasks. Finally, the outputs of Zelda Rose are *Transformers*-compatible models, ready to be loaded in this library or uploaded to HuggingFace Hub for immediate distribution.

3.2 Internal organization

The library is organized around two main building blocks: *tasks* and *datasets*.

Datasets are managers for collections of samples, e.g. raw sentences or parallel sentences. They contain the logic to load (from local files or remote repositories), preprocess (including at least tokenization, digitization and batching) and serve batch of samples to the training modules. In order to process large datasets with a limited memory footprint and enable caching, the data management and processing parts currently use of HuggingFace's *Datasets* library (Lhoest et al., 2021), wrapped in regular Pytorch and Pytorch-Lightning objects to make this transparent to the rest of the code and reduce the cost of changing this in future extensions of Zelda Rose.

Tasks are abstractions for the process of — given a model architecture and a configuration —, providing an object (in practice a Pytorch-Lightning training module) that implement the actual training process: generating targets from the inputs (for self-supervised tasks), obtain parameters for the optimization algorithm, run training steps, compute losses and metrics… In practice, since the models used are those implemented in HuggingFace's *Transformers* library (Wolf et al., 2020), which handles the forward pass and the loss computations, most tasks only have to specify target generation, optimization and monitoring-related parts.

Finally, given a configuration, the main module loads the appropriate task and dataset, to which it delegates all task- and data-related aspects, while taking care of the platform considerations, such as how many training processes to spawn, which devices to use and how many sample should they each process at once, etc. according to the configuration. This architecture allows for a clear separation of concerns, making adding new tasks and datasets quite easy. In practice, this is implemented by having the main module build a Pytorch-Lightning Trainer, which natively deals with a large number of hardware and training strategies and is actively maintained to follow the state of the art, in turn allowing us to benefits from the latest innovations at a relatively low maintenance cost.

4 Challenges

The main challenge with such a library is its maintainability given a limited time budget. Indeed, while the systematic reliance on third-party libraries rather than ad-hoc code as often as possible lets us benefit from their latest improvement with very little engineering code *a priori*, it also makes Zelda Rose dependent upon them for correctness and backward compatibility. This means that every new release of a dependency has to be checked with great care, as it could be introducing new bugs and it is often the case that slight but significant behavior changes go unnoticed or undocumented.

Undocumented behavior and implementation details in the dependencies were also a burden for the initial engineering effort and for subsequent extension of Zelda Rose (for instance when adding new tasks). Indeed, for libraries of these sizes and providing so many functionality, the documentation does not always follow the speed of bleeding edge evolutions, and checking their source code directly is often necessary to ensure that we use their interfaces correctly.

In other cases it was the lack of support for certain features required for the reproduction of existing work (such as embedding tying in the ELECTRA and DeBERTa models) that made reaching into private interfaces and monkey-patching necessary. While the dynamic nature of these libraries allow this, it makes part of our code much more brittle than we would hope for and these parts have to be checked for correctness with each new release.

This regular checking of non-regressions is made harder by the nature of the tool. Since train-

[1] https://huggingface.co/models

ing neural networks is not in general deterministic and since bugs tend to manifest not as outright errors but as degradation in performances, ensuring that a modification did not introduce a bug can be challenging. Indeed, it is often the case that such a degradation would not be observed on toy examples but only on the larger scale of real world examples. However, continuously running automated tests (in a continuous integration pipeline for instance) at these scales is not realistic, reducing the safety provided by tests.

Moreover, the complexity of the dependencies makes unit testing challenging, since it would often require mocking internals of these libraries. Automated tests ran as part of the continuous integration pipeline of Zelda Rose are therefore limited to so-called smoke tests, which verify if the tool runs, is able to train model in a variety of configuration and produces viable output. Non-regression tests, in contrast, are run manually, by training a few select models in realistic conditions. Since these are much more costly, they are only ran before a new version of Zelda Rose is released.

5 Conclusion

Zelda Rose makes possible it to pretrain transformer models with very little effort, even in complex environments. No custom code is needed to train a model on new data, which should allow more people to participate in resource development efforts. However, the modular design of this tool also makes it easy to extend and to integrate future developments in transformer models.

Building upon high level state-of-the-art frameworks rather than custom engineering allows to benefit from the latest innovations without too much maintenance work. It is not without downsides as it still requires some work to ensure that new releases of these dependencies do not introduce bugs and it makes the parts of Zelda Rose more complex to test individually. Overall, however, this design choice is still a net gain.

Planned future development in Zelda Rose will focus on adding more tasks, which will be the occasion to make the design even more modular by allowing tasks to be external plugins. More efforts will also be made on the instrumentation and testing, to make the testing process more cumbersome and add as many checks as possible of the reproducibility of training results. Finally, since the aim of this tool is to be useful beyond its developers, future improvements will also in a large part be guided by the requests of users, which we hope will be numerous and relevant!

References

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *Proceedings of the 8th International Conference on Learning Representations*.

Samuel Colvin, Eric Jolibois, Hasan Ramezani, Adrian Garcia, Terrence Dorsey, and David Montague. 2023. Pydantic.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

William Falcon and The PyTorch Lightning team. 2019. PyTorch Lightning.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. AllenNLP: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 1–6, Melbourne, Australia. Association for Computational Linguistics.

Pengcheng He, Jianfeng Gao, and Weizhu Chen. 2021. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. DeBERTa: Decoding-enhanced BERT with disentangled attention. In *Proceedings of the 2021 International Conference on Learning Representations*.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. 2021. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference*

on Empirical Methods in Natural Language Processing: System Demonstrations, pages 175–184, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020. Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Minh Van Nguyen, Viet Dac Lai, Amir Pouran Ben Veyseh, and Thien Huu Nguyen. 2021. Trankit: A light-weight transformer-based toolkit for multilingual natural language processing. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, pages 80–90, Online. Association for Computational Linguistics.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

Rob van der Goot, Ahmet Üstün, Alan Ramponi, Ibrahim Sharaf, and Barbara Plank. 2021. Massive choice, ample tasks (MaChAmp): A toolkit for multi-task learning in NLP. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, pages 176–197, Online. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008, Long Beach, California. Curran Associates, Inc.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Liu Zhuang, Lin Wayne, Shi Ya, and Zhao Jun. 2021. A robustly optimized BERT pre-training approach with post-training. In *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, pages 1218–1227, Huhhot, China. Chinese Information Processing Society of China.