xLM: A Python Package for Non-Autoregressive Language Models

Dhruvesh Patel Durga Prasad Maram* Sai Sreenivas Chintha* Benjamin Rozonoyer
Andrew McCallum
University of Massachusetts Amherst
{dhruveshpate, dmaram, saisreenivas, brozonoyer, mccallum}@umass.edu

xlm-core, xlm-models Docs Video Code

Abstract

In recent years, there has been a resurgence of interest in non-autoregressive text generation in the context of general language modeling. Unlike the well-established autoregressive language modeling paradigm, which has a plethora of standard training and inference libraries, implementations of non-autoregressive language modeling have largely been bespoke making it difficult to perform systematic comparisons of different methods. Moreover, each non-autoregressive language model typically requires it own data collation, loss, and prediction logic, making it challenging to reuse common components. In this work, we present the `xLM` python package, which is designed to make implementing small non-autoregressive language models faster. With a secondary goal of providing a suite of small pre-trained models (through a companion `xlm-models` package) that can be used by the research community.

1 Introduction

Autoregressive language models (ARLMs), which generate text sequentially from left to right by adding one token at a time, are well established with a plethora of standard training and inference libraries (Wolf et al., 2020; OLMo et al., 2024). However, recently, there has been a resurgence of research interest in non-autoregressive text generation due to its potential for faster inference speeds and better generation quality for certain tasks. Unlike left-to-right generation, non-autoregressive text generation can be done in many ways, for example, using masked diffusion language models (Sahoo et al., 2024), Gaussian diffusion language models (Gulrajani and Hashimoto, 2023), insertion language models (Patel et al., 2025), edit-based language models (Havasi et al., 2025), etc. Moreover, each method typically requires its own data collation, loss, and prediction logic, making it challenging to reuse common components across different methods. The rapidly expanding landscape of these methods has led to many bespoke implementations, making it extremely difficult to compare them systematically. In this work, we present the `xLM` python package, which aims to provide a unified framework for developing and comparing small non-autoregressive language models. `xLM` uses Pytorch (Paszke et al., 2019) as the deep learning framework, Pytorch Lightning (Falcon and The PyTorch Lightning team, 2019) for training utilities, and Hydra (Yadan, 2019) for configuration management. `xLM` is designed to make implementing small non-autoregressive language models faster without sacrificing flexibility.

The rest of the paper is organized as follows. Section 2 discusses the core design principles of `xLM`. In section 3, we discuss how `xLM` serves a unique purpose in the landscape of LLM libraries. Section 4 presents a high-level overview of the three core components of `xLM`, followed by section 5 that provides a step-by-step demonstration of how one would implement a new language model using `xLM`. Finally, Section 6 presents a set of benchmarking results where we implement three models in `xLM` to reproduce known results.

2 Design Principles

`xLM` follows the principle of maximal independence. The core library provides access to a small number of shared components, which are designed to be model independent, and can be used by any kind of language model. Each model implementation lives in its own folder/package and is completely independent of other models. This allows researchers to keep their model code clean, self-contained, and easy to share. It also allows them to use their model outside the `xLM` framework without refactoring. Maximal independence is achieved by

*Equal contribution second authors.

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics
Volume 3: System Demonstrations, pages 445–456
March 24-29, 2026 ©2026 Association for Computational Linguistics

following design choices.

Composition over inheritance. The maximal independence is achieved by using composition over inheritance (Gamma et al., 1995), wherein the core components delegate model specific logic to the specific model instance. For example, as shown in Figure 1, the `DataModule` carries a collection of `DatasetManager` *instances*, one for each dataset, and the creation of dataloaders is delegated to the respective `DatasetManager` instance depending on the stage (train, val, test, or predict). Similarly, the `Harness` carries instances of `Model`, `LossFunction`, `Predictor`, to which it delegates the model specific logic for forward pass, loss computation, and generation respectively. Moreover, one can swap out one or more of the four components with a different but compatible implementation without having to change all four.

Copy over branching. This principle goes against the common wisdom of not copying code. However, in case of research codebases, copying reduces code complexity and helps increase the speed of development (Wolf et al., 2020). It naturally creates independence. Moreover, it also allows templatizing the process of creating a brand new model which helps rapid prototyping by humans as well as LLMs (Li et al., 2025).

Arbitrary code injection. Python is a highly dynamic language, which allows one to inject arbitrary code at runtime. In production and public facing codebases, this creates a security risk, but this flexibility is a boon for research codebases, as it allows rapid prototyping and experimentation. As we will discuss in section 4.3, Hydra (Yadan, 2019) provides a powerful mechanism to inject arbitrary code at runtime by allowing one to fill a specific *slot* with an instance of any class.

3 Related Work

Due to the rapid development of LLMs, there are many libraries for training (Wolf et al., 2020; von Werra et al., 2020; Lightning-AI, 2023; OLMo et al., 2024) and inference using auto-regressive LLMs (Kwon et al., 2023). On the other hand, there are only a handful of python libraries that support non-autoregressive sequence modeling like FairSeq (Ott et al., 2019), and AllenNLP (Gardner et al., 2017). Moreover, even these libraries are no longer actively maintained and do not support non-autoregressive language modeling. To the best of our knowledge, `xLM` is the only library that supports fast prototyping of small non-autoregressive language models, and is geared towards providing a suite of reference implementations for up and coming non-autoregressive language modeling methods.

4 Core Components of `xLM`

In this section, we will discuss the the `Harness`, the `DataModule`, and the configuration management, which together handle the execution flow of all the supported workflows (section 5.7) like training, evaluation, prediction and debugging. In most use cases, these components need not be touched by the user, removing the need for most of the boilerplate code. [1]

4.1 DataModule

The base `TextDataModule`, which builds on top of Lightning DataModule, provides a generic, model and task agnostic interface for managing arbitrary number of text datasets. [2] This is achieved by using one `DatasetManager` per dataset as shown in fig. 1. Each `DatasetManager` instance is responsible for managing the complete lifecycle of a single dataset, including downloading, preprocessing, caching and managing the data collator and data-loader options. It has slots for the following components that allow injecting custom logic:

- `Dataset` which could point to a HuggingFace dataset or a custom dataset.
- `Collator` and `Preprocessor`, both of which can depend on the model type as well as the task.

Complete flexibility and independence is achieved by allowing a many-to-many mapping between the `DatasetManager` instances and the workflow stages (train, val, test, predict), wherein a single `DatasetManager` instance can be mapped to multiple workflow stages, and vice versa. This ensures that only a single copy of the dataset is loaded into memory but if needed it can be used at multiple places in the workflow.

The core implementation of `DatasetManager` supports all common training strategies for small models: single-node single-GPU, single-node multi-GPU and multi-node multi-GPU, with map style and iterable style dataset support for each.

[1] `xLM` also has some useful additional features described in appendix F.

[2] See `https://lightning.ai/docs/pytorch/stable/data/datamodule.html`. The interface is not specific to text datasets, and can be used for any kind of sequence datasets.

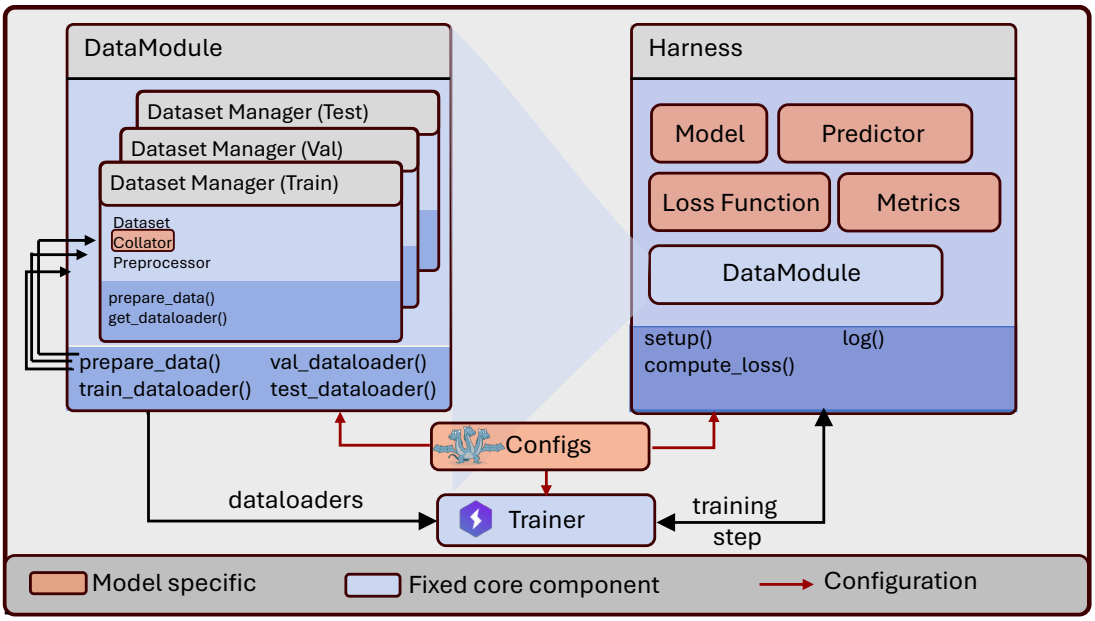

Figure 1: Overview of xLM design. It consists of two classes of components: the core components (Harness and DataModule) and the model-specific components, whose implementations depend on the model logic. These components are defined in the configuration files managed by Hydra (see fig. 3), enabling arbitrary component swapping. The Harness component is responsible for instantiating all components (model, loss, predictor, etc.) and delegating their respective functionalities. The DataModule component manages multiple datasets across workflow stages using DatasetManager objects, each handling a dataset and an appropriate Collator.

The user simply needs to provide the respective arguments in the config file.

4.2 Harness

The Harness is the main class that inherits from the PyTorch lightning's LightningModule [3], and is responsible for instantiating all the components like the model, loss function, predictor, etc., based on the configuration files. As shown in fig. 1, the Harness has slots (attributes) for all the core components, and it delegates the model specific logic to the respective components' methods. Inheriting from the LightningModule allows us to use all the features of PyTorch lightning, such as logging, checkpointing, saving, etc., and also allows us to use the LightningTrainer. See appendix C for more details.

4.3 Configuration Management

In xLM, the configuration files have two roles. First, like any other configuration system (e.g. Python's `ArgParse`), it allows the user to specify various parameter values that dictate the behavior of the system. Second, through the use of `hydra.utils.instantiate`, it allows swapping out entire components directly from the configuration file, without changing a single line of python code. This enables arbitrary code injection at runtime. Hydra configs themselves can be arbitrarily nested, and one config file can be referred in another config file, the composition of which is automatically taken care of by Hydra. This enables a modularization of the configuration files themselves.[4]

5 Demonstration

In this section, we will walk through, step by step, the process of implementing a new language model using the xLM library. In order to demonstrate the flexibility of the library, we pick a non-standard language modeling paradigm for this demonstration. Specifically, we will implement the Insertion Language Model (ILM) of Patel et al. (2025), which generates text by iteratively inserting tokens in the existing sequence by selecting the position and the vocabulary item to insert. In order to keep the demonstration simple, we will use the synthetic seq2seq task of generating a path on a star shaped graph as the task (Patel et al., 2025) on the StarEasy dataset [3] . We also provide a demonstration of training ILM on the LM1B corpus (Chelba et al., 2014) in Appendix A.

[3] https://lightning.ai/docs/pytorch/stable/common/lightning_module.html

[4] Please refer to the Hydra documentation for more details https://hydra.cc/docs/intro/.

```
ilm/ ..................................... model root directory
├── __init__.py
├── types_ilm.py ................................ type definitions
├── model_ilm.py ................................ neural network
├── loss_ilm.py ..................................loss function
├── predictor_ilm.py ............................ inference logic
├── datamodule_ilm.py ..............................collators
├── metrics_ilm.py ..................... metrics implementation
├── configs/ ..............................hydra configurations
│   ├── model/
│   │   └── ilm.yaml ............................ neural net params
│   ├── model_type/
│   │   └── ilm.yaml ..................... loss, metrics, predictors etc.
│   ├── collator/
│   │   ├── default_ilm.yaml
│   │   ├── seq2seq_ilm.yaml
│   │   └── seq2seq_pred_ilm.yaml
│   ├── datamodule/
│   │   └── star_easy_ilm.yaml.....................dataset managers
│   └── experiment/
│       └── star_easy_ilm.yaml
├── setup.py ..........................................installation
└── README.md ..................................... documentation
```

Figure 2: Directory structure generated by the scaffolding script.

To construct a new model, we need to create a fresh directory for the model, which will contain the implementations of the model specific components: `Model`, `LossFunction`, `Predictor`, and `Collators`. It will also contain their respective configuration files. In order to make the process of creating a new model easier, we provide a scaffolding script that automatically generates the necessary files. It can be invoked by executing `xlm-scaffold ilm` which will generate the directory structure as shown in fig. 2.

The scaffold files already contain placeholder empty class and function declarations. Next we will walk through the process of adding implementation for each of the python files shown in fig. 2. Finally, we will show how to reference these components configuration files.

5.1 Model

In `model_ilm.py`, we define the neural network backbone for the model which inherits from `torch.nn.Module` and implements the `forward()` method. We have the complete freedom to decide the arguments that the `forward()` method takes.

```python
from xlm.model import Model
from xlm.modules.rotary_transformer import
    RotaryTransformerLayer

class ILMModel(torch.nn.Module, Model):
    ...
    def forward(self, input_ids, attention_mask,...):
        self.encoder_layer = RotaryTransformerLayer(
            d_model,
            nhead,
            dim_feedforward,
            dropout,
            activation,
            layer_norm_eps,
        )
        ...
        return vocab_logits, stopping_logits
```

Many of the `xLM` modules detailed in section F.1 can be used for building the architecture like the use of `RotaryTransformerLayer` here for the encoder. After defining the model class, the constructor arguments needed for default instantiation along with the fully qualified class path are stored in the config `configs/model/ilm.yaml` file:

```yaml
# @package _global_
model:
  _target_: ilm.model_ilm.ILMModel
  ...
```

5.2 LossFunction

The `LossFunction` is a callable that takes in a batch of inputs and returns a dictionary of values, with a mandatory "loss" key and any other optional values that we want to log. In ILM, the loss function computes two components: (1) a cross-entropy loss over only the positions where tokens were dropped, and (2) a binary classification loss that predicts whether input sequence is complete.

```python
from xlm.harness import LossFunction
from types_ilm import ILMBatch, ILMLossDict

class ILMLoss(LossFunction[ILMBatch, ILMLossDict]):
    def loss_fn(self, batch: ILMBatch, ...) -> ILMLossDict:
        vocab_logits, stopping_logits = self.model(**batch)

        vocab_logit_loss = masked_cross_entropy(vocab_logits,
        batch["target_ids"])

        stopping_loss = binary_cross_entropy(stopping_logits)
        return {"loss": vocab_logit_loss + stopping_loss, ...}
```

The default loss parameters are stored in the config `configs/model_type/ilm.yaml` file under the `loss` key as shown in fig. 5.

5.3 Data Pipeline

To setup the data pipeline, we just need to configure the dataset and implement model specific collators for each dataset, and add the configuration files for the `DatasetManagers` and `Collators`. The `xLM` comes with some synthetic seq2seq, and language modeling datasets preconfigured. This includes the synthetic `StarEasy` dataset. [5] For this demonstration, we will use the preconfigured `StarEasy` dataset, wherein each example consists of a prompt and a target path. The prompt contains an edge list (in random order) of a star graph and the start and end nodes. The target contains the gold path from the start node to the end node (Patel et al., 2025). An example prompt and target is shown in fig. 4.

[5] See appendix D.1 for learning how to configure a new dataset, and appendix G for the list of preconfigured tasks.

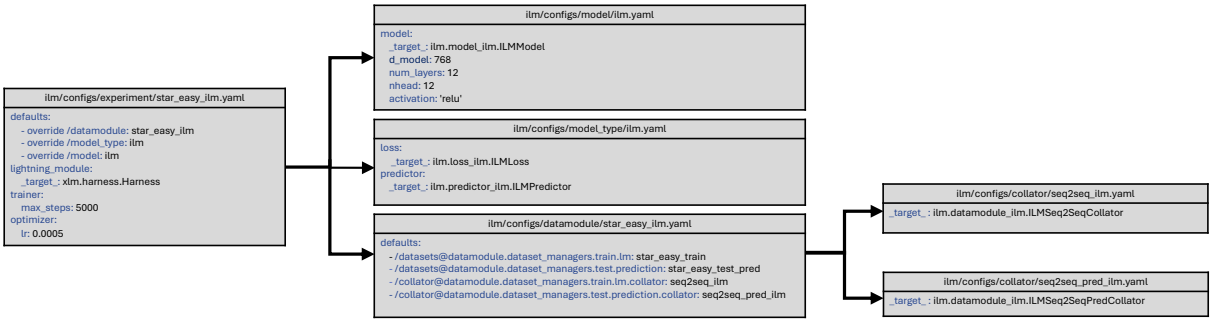

Figure 3: Configuration tree for a typical experiment (e.g. for ILM for a seq2seq planning task on the `StarEasy` dataset). The `experiment` config is at the root of the nesting structure, contains global parameters, and composes component configs (`model`, `model_type`, and `datamodule`). The `model/ilm.yaml` file stores the parameters for the model class. The `model_type/ilm.yaml` file contains the information needed to instantiate the loss function, predictor, and metric components. The `datamodule/star_easy_ilm.yaml` composes the configs of the `DatasetManagers` and `Collators` (here, `StarEasy` and seq2seq collators). Note: Only partial entries are shown in the figure for brevity.

```
edge_list = [[1, 5], [1, 7], [7, 9]]
source = 1
goal = 9
path = [1, 7, 9]
sequence = "CLS 1 7 1 5 7 9 1 9 BOS 1 7 9 PAD PAD"
```

Figure 4: An example of a prompt and target for the `StarEasy` dataset.

5.4 Collators

A `Collator` is a callable that takes in a list of raw examples from the dataset and returns a batch to be fed to the `LossFunction` or the `Predictor`. Typically, a `Collator`'s implementation depends on the model type and the type of task, e.g. the collator for a seq2seq task will be different from that of unconditional language modeling task. Moreover, a collator for loss computation will be different from that used for prediction. For the synthetic seq2seq task of star graphs, we need to implement two collators: one for training and one for prediction. The training collator randomly drops tokens from the gold path and places the dropped tokens under the `"target_ids"` key in the batch. The prediction collator only keep the prompt under the `"input_ids"` key in the batch leaving all the tokens in the path to be predicted by the model.

```
from xlm.datamodule import Collator
from types_ilm import ILMBatch

class ILMSeq2SeqCollator(Collator):
    def __call__(self, examples: List) -> ILMBatch:
        prefix = prepare_prefix_ids([e["prompt_ids"] for e in
        examples])
        suffix = drop_tokens([e["input_ids"] for e in
        examples])
        return {
            "input_ids": torch.cat([prefix["input_ids"],
            suffix["input_ids"]], dim=1),
            "target_ids": suffix["target_ids"],
            "n_drops": suffix["n_drops"]
        }
```

```
class ILMSeq2SeqPredCollator(ILMSeq2SeqCollator):
    def __call__(self, examples: List) -> ILMBatch:
        prefix = prepare_prefix_ids([e["prompt_ids"] for e in
        examples])
        target_ids =
        prepare_target_ids_for_test([e["input_ids"] for e in
        examples])
        return {
            "input_ids": prefix["input_ids"],
            "target_ids": target_ids["target_ids"],
            "n_drops": None
        }
```

The collators are configured by adding `configs/collator/seq2seq_ilm.yaml`:

```
_target_: ilm.datamodule_ilm.ILMSeq2SeqCollator
```

and `configs/collator/seq2seq_pred_ilm.yaml`:

```
_target_: ilm.datamodule_ilm.ILMSeq2SeqPredCollator
```

Finally, the entire data pipeline is configured by adding `configs/datamodule/star_easy_ilm.yaml` config file as shown in fig. 6.

5.5 Predictor

The `Predictor` is a callable that implements the inference loop for the model. In ILM, the `Predictor` implements iterative infilling by repeatedly sampling the location of insertion and the vocabulary item to insert till the stopping classification head indicates that generation should stop.

```
from xlm.harness import Predictor
from types_ilm import ILMBatch,ILMPredictionDict

class ILMPredictor(Predictor[ILMBatch, ILMPredictionDict]):
    def predict(self, batch: ILMBatch, ...) ->
    ILMPredictionDict:
        ...
        return {"text": decoded_texts, "history":
        generation_history}
```

xLM automatically invokes the predictor during evaluation and handles decoding and metric computation. The predictor class path is stored in the

config `configs/model_type/ilm.yaml` file under the `predictor` key as shown in fig. 5

5.6 Experiment Configuration

Once the individual components are implemented and configured as elaborated previously, the hierarchical configuration tree is formed through the *experiment config*. Figure 3 depicts the configuration tree for the ILM model on the `StarEasy` dataset. The arrows in the figure indicate the nesting structure: the main experiment config `experiments/star_easy_ilm.yaml` refers to the `model` `model/ilm.yaml`, `model_type` `model_type/ilm.yaml`, and the `datamodule` `datamodule/star_easy_ilm.yaml`, which in turn refers to the the collators.

5.7 Workflows

Model Discovery After creating the experiment configuration, we are ready to use the model. However, before we can do that we need to make the model *discoverable*. There are two ways to do that:

1. **Python Package**: The model can be installed as a standalone python package, followed my setting the environment variable `XLM_MODELS_PACKAGES` as ":" separated list of installed model packages, e.g. `XLM_MODELS_PACKAGES=ilm:mlm`.
2. **Directory**: Alternatively, one can simply set `XLM_MODELS_PATH` to point to the parent directory that contains the model folder, e.g. if the model is located at `/path/to/ilm`, then `XLM_MODELS_PATH=/path/to`.

Once the model is discoverable, `xLM` provides three main workflows: training, evaluation, and generation. They can be run using the following command

```
$ xlm job_type=[JOB_TYPE] job_name=[JOB_NAME]
experiment=[CONFIG_PATH]
```

The `job_type` argument can be one of `train`, `eval` and `generate`. The name of the experiment config file (without the `.yaml` extension) containing all necessary overrides is given under the `experiment` option. `xLM` also provides a group of debug settings that can be used to perform a quick debug run that tries to overfit on a single batch of data. This can be used by appending the `debug=overfit` option to the command. Finally, the model code can be packaged as a standalone python package.

Table 1: Benchmark performance on planning seq2seq task on star graphs. The columns represent token and sequence accuracies for each model.

Model	Easy		Medium		Hard	
	Seq.	Token	Seq.	Token	Seq.	Token
ARLM	33.1	81.7	77.2	82.1	25.2	43.7
ILM	100.0	100.0	100.0	100.0	97.5	98.2
MLM	100.0	100.0	83.1	98.0	25.3	79.6
MDLM	100.0	100.0	36.5	90.6	21.0	54.9

```
# Packaging
python setup.py sdist bdist_wheel
# Installation
pip install ilm
```

Post training, the model weights can be extracted and uploaded to the Hugging Face model hub (see appendix F for the details).

6 Benchmarks

`xLM` is a framework aimed at making research prototyping of small non-autoregressive language models easier. The purpose of this section is to show that the library can be used to reproduced known results for small non-autoregressive language models. As representative tasks, we pick one seq2seq task and one unconditional language modeling task. Specifically, we reproduce the results on the synthetic seq2seq of path finding on star graphs and unconditional language modeling tasks on LM1B.

Synthetic Planning Tasks We use the hyperparameters reported in for training auto-regressive model (ARLM), masked diffusion model (MDLM)(Sahoo et al., 2024), masked language model (MLM) and insertion language model (ILM) on the three variants of the synthetic planning tasks (Patel et al., 2025). The results (table 1) are within 2% of the reported results in the original papers.

Language Modeling We train a 12 layer transformer as ARLM, MDLM, and ILM, respectively, on the LM1B corpus (Chelba et al., 2014). In order to make the results comparable, we use negative loglikelihood under Llama 3.2 8B model as the metric. The results are reported in table 2, which are close to the results reported in Patel et al. (2025) for the same settings.

7 Conclusion and Future Work

We presented a modular python package that makes prototyping small non-autoregressive language models easier. We plan on adding refer-

Table 2: Benchmark performance on unconditional language modeling on LM1B. The rows represent negative loglikelihood (under Llama 3.2) and entropy of the generated sequences for each model.

	corpus	ARLM	MDLM	ILM
NLL	3.71	3.94	4.81	4.72
Entropy	3.08	3.12	3.70	2.81

ence implementations and benchmark more models (Havasi et al., 2025) as well as add support for non-text sequence generation tasks like molecule generation (see appendix H for details).

References

Gregor Bachmann and Vaishnavh Nagarajan. 2024. The pitfalls of next-token prediction. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 2296–2318. PMLR.

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2014. One billion word benchmark for measuring progress in statistical language modeling. *Preprint*, arXiv:1312.3005.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. Flex attention: A programming model for generating optimized attention kernels. *Preprint*, arXiv:2412.05496.

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

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., USA.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. Allennlp: A deep semantic natural language processing platform.

Ishaan Gulrajani and Tatsunori Hashimoto. 2023. Likelihood-based diffusion language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Marton Havasi, Brian Karrer, Itai Gat, and Ricky T. Q. Chen. 2025. Edit Flows: Flow Matching with Edit Operations. *Preprint*, arXiv:2506.09018.

John J Irwin and Brian K Shoichet. 2005. Zinc- a free database of commercially available compounds for virtual screening. *Journal of chemical information and modeling*, 45(1):177–182.

Jaeyeon Kim, Lee Cheuk-Kit, Carles Domingo-Enrich, Yilun Du, Sham Kakade, Timothy Ngotiaoco, Sitan Chen, and Michael Albergo. 2025. Any-Order Flexible Length Masked Diffusion. *Preprint*, arXiv:2509.01025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Pengfei Li, Qichang Zheng, and Ziyi Jiang. 2025. An empirical study on the accuracy of large language models in api documentation understanding: A cross-programming language analysis. *Journal of Computing Innovations and Applications*, 3(2):1–14.

Lightning-AI. 2023. Litgpt. `https://github.com/Lightning-AI/litgpt`.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, and 21 others. 2024. 2 olmo 2 furious. *Preprint*, arXiv:2501.00656.

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.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, and 2 others. 2019. Pytorch: An imperative style, high-performance deep learning library. *Preprint*, arXiv:1912.01703.

Dhruvesh Patel, Aishwarya Sahoo, Avinash Amballa, Tahira Naseem, Tim GJ Rudner, and Andrew McCallum. 2025. Insertion language models: Sequence generation with arbitrary-position insertions. *arXiv preprint arXiv:2505.05755*.

William Peebles and Saining Xie. 2023. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4195–4205.

Lars Ruddigkeit, Ruud Van Deursen, Lorenz C Blum, and Jean-Louis Reymond. 2012. Enumeration of 166 billion organic small molecules in the chemical universe database gdb-17. *Journal of chemical information and modeling*, 52(11):2864–2875.

Subham Sekhar Sahoo, Marianne Arriola, Aaron Gokaslan, Edgar Mariano Marroquin, Alexander M. Rush, Yair Schiff, Justin T. Chiu, and Volodymyr Kuleshov. 2024. Simple and Effective Masked Diffusion Language Models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. Trl: Transformer reinforcement learning. `https://github.com/huggingface/trl`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Huggingface's transformers: State-of-the-art natural language processing. *Preprint*, arXiv:1910.03771.

Omry Yadan. 2019. Hydra - a framework for elegantly configuring complex applications. Github.

A Demonstration on LM1B

In section 5, we demonstrated ILM on a seq2seq task where the model generates a path given a graph description as a prompt. Here, we show how to implement ILM for unconditional language modeling on a real-world text corpus (LM1B), where there is no prompt and the model generates text from scratch. The steps translate to any other type of language model.

A.1 Collator for Unconditional LM

The first difference from the seq2seq task is the collator. In the case of ILM, instead of `ILMSeq2SeqCollator`, which protects the prompt tokens from being dropped, we need to implement a new collator, which we call `DefaultILMCollator`, which drops tokens uniformly from the entire sequence:

```python
from xlm.datamodule import Collator

class DefaultILMCollator(Collator):
    """Used for pre-training."""

    def __call__(self, examples: List) -> ILMBatch:
        batch = ilm_single_segment_collate_target_fn(
            [e["input_ids"] for e in examples],
            ...
            sample_n_drops_fn=_n_drop_uniformly,  # uniform
            drop count
            drop_indices_fn=_drop_uniformly,      # uniform
            drop positions
        )
        return batch
```

A.2 Datamodule Configuration

The datamodule configuration for unconditional language modeling differs from the seq2seq task at two places: (1) we need to swap out the collator configuration to use our new `DefaultILMCollator`, and (2) we need to add a new dataset manager for unconditional generation, which manages dataset of blank sequences.

```yaml
# configs/datamodule/lm1b_ilm.yaml
# @package _global_
defaults:
  - /datasets@datamodule.dataset_managers.train.lm:
  lm1b_train
  - /datasets@datamodule.dataset_managers.val.lm: lm1b_test
  - /datasets@datamodule.dataset_managers.val.unconditional
  _prediction:
      text_unconditional_prediction
  - /collator@datamodule.dataset_managers.train.lm.collator:
  default_ilm
  - /collator@datamodule.dataset_managers.val.lm.collator:
  default_ilm
  - /collator@datamodule.dataset_managers.val.unconditional
  _prediction.collator:
      default_ilm
```

For unconditional generation, the model must generate text starting from an empty sequence. This is handled by `ILMEmptyDataset`, which generates empty examples that the model fills entirely:

```python
class ILMEmptyDataset(IterableDataset):
    def __init__(self, tokenizer: Tokenizer, num_examples:
    int):
        self.tokenizer = tokenizer
        self.num_examples = num_examples

    def __iter__(self):
        for _ in range(self.num_examples):
            yield self.tokenizer("", add_special_tokens=False)
```

This dataset is referenced in the experiment configuration:

```yaml
# configs/experiment/lm1b_ilm.yaml
datamodule:
  dataset_managers:
    val:
      unconditional_prediction:
        dataset_constructor_str:
        ilm.datamodule_ilm.ILMEmptyDataset
```

B Hydra Configs

Figure 5 and fig. 6 and show the model type and datamodule configs, respectively.

C Harness

This section describes the functionality implemented in `Harness`.

```yaml
# @package _global_
loss:
  _target_: ilm.loss_ilm.ILMLoss
  ... # other arguments

predictor:
  _target_: ilm.predictor_ilm.ILMPredictor
  ... # other arguments

reported_metrics:
  train: # reported during training loop
    lm: # dataloader name
      accumulated_loss: # metric name
        prefix: train/lm # str prefix for logging
        update_fn: ilm.metrics_ilm.mean_metric_update_fn #
        callable
    ... # metrics for additional dataloaders
  val:
    ...
  test:
    ...
```

Figure 5: The `configs/model_type/ilm.yaml` config file for the ILM model. It contains sections for `LossFunction`, `Predictor` and `Metrics`.

C.1 Loggers

Logger components can be registered through the `loggers` key. `xLM` provides pre-configured tensorBoard and WandB logger configurations, with tensorboard being the default. However, all PyTorch Lightning–supported loggers can also be used.

```yaml
defaults:
  - override /loggers:
    -wandb
```

Figure 7: The entries for loggers in the `experiment` config file.

C.2 Logging Metrics

The library provides various preconfigured metrics for different stages, such as **accumulated loss** (mean loss value), **exact match**, and **token accuracy**. Each of these metric components inherits from the `torchmetrics.Metric` class and is wrapped by default using the `xlm.metrics.MetricWrapper` module, which manages the computation of its value. Another key method to define is the `update_fn` function, which takes raw input batch sequences and loss function outputs, and transforms them into a dictionary of entries used by the Metric class to compute the final value. This allows for customization, enabling custom metric logic depending on the model and task. Different metrics can be configured for various workflow stages as depicted in fig. 8.

```yaml
# @package _global_
defaults:
  - /datasets@datamodule.dataset_managers.train.lm:
  star_easy_train
  - /datasets@datamodule.dataset_managers.val.lm:
  star_easy_val
  - /datasets@datamodule.dataset_managers.val.prediction:
  star_easy_val_pred
  - /datasets@datamodule.dataset_managers.test.lm:
  star_easy_test
  - /datasets@datamodule.dataset_managers.test.prediction:
  star_easy_test_pred
  -
  /datasets@datamodule.dataset_managers.predict.prediction:
  star_easy_test_pred
  - /collator@datamodule.dataset_managers.train.lm.collator:
  seq2seq_ilm
  - /collator@datamodule.dataset_managers.val.lm.collator:
  seq2seq_ilm
  - /collator@datamodule.dataset_managers.val.prediction.co
  llator:
  seq2seq_pred_ilm
  - /collator@datamodule.dataset_managers.test.lm.collator:
  seq2seq_ilm
  - /collator@datamodule.dataset_managers.test.prediction.c
  ollator:
  seq2seq_pred_ilm
  - /collator@datamodule.dataset_managers.predict.predictio
  n.collator:
  seq2seq_pred_ilm
...
```

Figure 6: The `configs/datamodule/star_easy_ilm.yaml` config file for the ILM model. It contains sections for datasetmanagers and collators.

C.3 Logging Predictions

The model predictions for validation and test sets are logged under the `logs/runs` directory. The configuration for this is specified using the `log_predictions` key, and `xLM`'s `xlm.log_predictions.LogPredictions` component should be used. The logger file contains a **text** field, which contains the prefix and generated sequence, and a **truth** field, which contains the ground-truth sequence. Predictions can be logged to a local file, trainer loggers, or the console by using the values 'file', 'logger', or 'console'.

```yaml
log_predictions:
  _target_: xlm.log_predictions.LogPredictions
  fields_to_keep_in_output:
    - text
    - truth
  inject_target: target_ids
  writers:
    - file
    - logger
```

Figure 9: The entries for logging predictions in the `experiment` config file.

```yaml
reported_metrics:
  train:
    lm:
      accumulated_loss:
        _target_: xlm.metrics.MetricWrapper
        name: accumulated_loss
        metric:
          _target_: torchmetrics.MeanMetric
        prefix: train/lm
        update_fn:
        xlm.lm.ilm.metrics_ilm.mean_metric_update_fn
  val:
    ...
  test:
    lm:
      accumulated_loss:
        _target_: xlm.metrics.MetricWrapper
        name: accumulated_loss
        metric:
          _target_: torchmetrics.MeanMetric
        prefix: test/lm
        update_fn:
        xlm.lm.ilm.metrics_ilm.mean_metric_update_fn
    prediction:
      exact_match:
        _target_: xlm.metrics.MetricWrapper
        name: exact_match
        metric:
          _target_: xlm.metrics.ExactMatch
        prefix: test/prediction
        update_fn: xlm.metrics.seq2seq_exact_match_update_fn
      token_accuracy:
        _target_: xlm.metrics.MetricWrapper
        name: token_accuracy
        metric:
          _target_: xlm.metrics.TokenAccuracy
        prefix: test/prediction
        update_fn:
        xlm.metrics.seq2seq_token_accuracy_update_fn
```

Figure 8: Metric related entries in `configs/model_type/ilm.yaml` config file for the ILM Model.

D FAQs

D.1 How to add a new task/dataset?

To add a new task, one must prepare the corresponding datasets in a Hugging Face (HF) dataset compatible format. Depending on the use case, separate dataset configuration (.yaml) files can be created for each stage (train/val/test/pred), providing the flexibility to process the same dataset differently or to use entirely different datasets across stages. The HF-downloadable dataset can be specified using the `full_name` key, and the `xlm.datamodule.DatasetManager` from xLM can be reused for dataset manager instantiation. This manager automatically handles downloading, caching, preprocessing, and data-loading operations according to the dataset configuration entries. Alternatively, datasets can be used locally without being uploaded to the HF Hub by employing `xlm.datamodule.LocalDatasetManager`. Once these configuration files are prepared, they must be registered in the `configs/datamodule/MODEL.yaml` (Figure 3).

D.1.1 Implementing custom `DatasetManager`

In addition to xLM's datasetmanager component, one can implement a custom datasetmanager for greater flexibility by inheriting from it. The necessary methods can be overridden—for example, the `_download` method shown below, where custom logic can be added to read and process the required type of file format.

```python
from xlm.datamodule import DatasetManager
import datasets

class CustomDatasetManager(DatasetManager):

    def _download(self) -> datasets.Dataset:
        ...
        return dataset
```

The component can be configured along with mentioning the necessary arguments by adding `configs/datasets/custom_dataset_train.yaml`

```yaml
_target_: CustomDatasetManager
... # other arguments
```

This config can then be registered in the `configs/datamodule/MODEL.yaml` file

```yaml
defaults:
-/datasets@datamodule.dataset_managers.train.lm:
custom_dataset_train
... # other dataset and collator entries
```

E Troubleshooting

E.1 Hydra Errors

`Unable to find a package ...` error by Hydra: See the name of the package in the error message. For example, if you encounter `Unable to find or instantiate abc.xyz.MyClass`, first try to import it manually in the Python interpreter: `python -c "from abc.xyz import MyClass"`.

Hydra Composition Errors: First check the Hydra documentation https://hydra.cc/docs/intro/. If the error persists, write a single experiment config without using defaults list for components.

F Additional Features

F.1 Modules

The library, under the *models* component, provides several architectural implementations that can be used to easily build diverse model backbones for prevalent non-autoregressive workflows in the literature. We provide modules for standard decoder

only transformer, Diffusion Transformers (Peebles and Xie, 2023) , rotary embedder (Su et al., 2024), time embedder, adaptive layer normalization layers (Peebles and Xie, 2023, and some standard noise schedulers. They are available under `xlm.modules`.

F.2 Push to hub

The library provides a `push_to_hub` command that uploads trained models to the Hugging Face Hub by reconstructing the complete training environment (datamodule, tokenizer and model architecture) from Hydra configurations.

```
$ xlm-push-to-hub experiment=[CONFIG_PATH]
+hub_checkpoint_path=[CKPT_PATH] +hub.repo_id=[HUB_PATH]
```

The environment variable `HF_HUB_KEY` must be assigned a valid Hugging Face access token.

F.3 Callbacks

The library extends the PyTorch Lightning callback infrastructure, enabling modular components to integrate with the training cycle (e.g., per-batch updates, validation hooks) in a decoupled manner. It provides a set of extensible callbacks, such as an Exponential Moving Average callback `EMACallback` for maintaining smoothed evaluation weights, a Checkpoint Monitoring callback `ModelCheckpoint` and a Performance Monitoring callback `SpeedMonitorCallback` for tracking training speed and identifying bottlenecks. The callback config file names (`xLM`'s or custom callbacks) can be mentioned in the following way to override the default callbacks:

```
defaults:
  - override /callbacks:
    - ema
    - speed_monitor
    - checkpoint_monitor
```

F.4 Checkpointing

The library provides a checkpointing system that saves training state (model weights, optimizer state, and training progress) to enable recovery from failures and long-running training jobs. It integrates with PyTorch Lightning's built-in `ModelCheckpoint` to save the best-performing model. It also supports frequent lightweight checkpoints using a `ThinningCheckpoint` callback that retains only milestone intervals to save storage and an `OnExceptionCheckpoint` callback that preserves state during crashes.

G Preconfigured Tasks and Models

Star Graphs The library provides three synthetic datasets that involve generating the path from a designated start node to a target node on star-shaped graphs (Bachmann and Nagarajan, 2024; Patel et al., 2025). The three variants follow the naming convention and construction of Patel et al. (2025). `StarEasy` contains symmetric graphs with the start node fixed at the junction, while `StarMedium` & `StarHard` introduce asymmetric structures with variable arm lengths and start nodes that may lie off the junction.

Language Modeling For text generation, we provide training and testing config setup for two datasets - LM1B, a large-scale corpus from the news domain, consisting of short text sequences (typically 2–3 sentences), and OpenWebText, which are widely used to benchmark the performance of small language models.

Models We benchmark and release three preconfigured models: ARLM, MDLM and ILM. [6]

H Planned Features

Non-text datasets Non-autoregressive sequence generation is useful for non-text tasks like molecule generation (Irwin and Shoichet, 2005; Ruddigkeit et al., 2012), path planning, etc. We plan on adding support for external non-text datasets in the future

New Models We plan to add support for newer models (Havasi et al., 2025; Kim et al., 2025).

FlexAttention Pytorch 2.5 introduces FlexAttention (Dong et al., 2024), dynamically compiled attention layer which allows fast attention with arbitrary masks. This can be very useful for non-autoregressive sequence generation as it can allow sequence packing eliminating the need for padding even for non-autoregressive models.

I Resources

Table 3 lists the resources provided through this work.

[6] We are working on benchmarking newer models, which will be released soon.

Table 3: Resources

Name	Type	Description	Link	License	Source
StarEasy	Dataset	Synthetic star graph dataset	https://github.com/facebookresearch/xlm/tree/main/xlm/datasets/star_easy	CC-BY-NC-SA	(Patel et al., 2025)
StarMedium	Dataset	Synthetic star graph dataset	https://github.com/facebookresearch/xlm/tree/main/xlm/datasets/star_medium	CC-BY-NC-SA	(Patel et al., 2025)
StarHard	Dataset	Synthetic star graph dataset	https://github.com/facebookresearch/xlm/tree/main/xlm/datasets/star_hard	CC-BY-NC-SA	(Patel et al., 2025)
xlm-core	Python Package	xlm-core package for non-autoregressive language modeling	https://pypi.org/project/xlm-core/	MIT	
xlm-models	Python Package	Companion package for xlm-core containing model implementations	https://pypi.org/project/xlm-models/	MIT	