







MAMMOTH

Massively Multilingual Modular Open Translation @ Helsinki

Timothee Mickus   Stig-Arne Grönroos   Joseph Attieh   Michele Boggia 
Ona De Gibert   Shaoxiong Ji   Niki Andreas Loppi  
Alessandro Raganato   Raúl Vázquez   Jörg Tiedemann 

 University of Helsinki  Silo AI
 University of Milano-Bicocca  NVIDIA
 Equal contribution  Alphabetical order

Abstract

NLP in the age of monolithic large language models is approaching its limits in terms of size and information that can be handled. The trend goes to modularization, a necessary step into the direction of designing smaller sub-networks and components with specialized functionality. In this paper, we present the MAMMOTH toolkit: a framework designed for training massively multilingual modular machine translation systems at scale, initially derived from OpenNMT-py and then adapted to ensure efficient training across computation clusters. We showcase its efficiency across clusters of A100 and V100 NVIDIA GPUs, and discuss our design philosophy and plans for future information. The toolkit is publicly available online.

 github.com/Helsinki-NLP/mammoth

1 Introduction

The field of NLP has recently witnessed a hastened transition towards ever-larger monolithic neural networks, exposed to gargantuan amounts of data so as to properly fit their humongous number of parameters. There is also a growing consensus that this is not a sustainable trend: This approach falters whenever data is scarce, and leads to costs—both financial and ecological—that cannot be disregarded.

The problems of scalability are especially prominent in the field of multilingual NLP. Scaling a multilingual model to a high number of languages is prone to suffer from interference, also known as the curse of multilinguality, leading to degradation in per-language performance, mainly due to the limited model capacity (Conneau et al., 2020; Wang et al., 2020). Increasing the overall model size, on the other hand, hits the ceiling in terms of trainability limited by hardware, data and training

algorithms. Modularity is one approach attempting to answer the challenges of scalability.

What is modularity? Modularity can be viewed in two complementary ways: as sparsity or as conditional computation. In the former, modularity enforces a principled sparsity of the network, so as to allow a model to be large at training time, but small during inference. In the latter, a modular approach entails routing the information flow to a module in order to select specific model parameters to be used in specific circumstances.

A canonical example is the use of language-specific encoders and decoders for machine translation: For the translation direction going from a source language L_S to a target language L_T , one would use an encoder trained on the data from the L_S source language, and a decoder trained to handle any and all inference to the L_T target language. Note, that the encoder will typically have seen data with L_S as source language but other languages than L_T as target languages, and mutatis mutandis for the decoder. This dynamic selection of modules entails a principled sparse activation: In this translation scenario, any encoder or decoder linked to some third language L_Z would not contribute to the computation, as if was set at zero.

This design can therefore lead to more efficient inference, since we can avoid computations for a large proportion of the parameters. The modularity also aids in the interpretability of parameters, as it is easy to determine which tasks a parameter contributes to. It also fosters the design of reusable neural network components: Modules can in principle be combined to allow for zero-shot adaptations to novel tasks and situations (Pfeiffer et al., 2021).

What we provide. One of the challenges that come with the study of modularity is the lack of a consensual, broadly available framework for de-

signing and handling such models. The MAMMOTH toolkit is meant to address this gap in the current NLP ecosystem. We build upon the OpenNMT-py library (Klein et al., 2018) and provide a set of utilities to effectively train modular encoder-decoder systems. The MAMMOTH toolkit is tailored toward efficient computation across clusters of compute nodes, and covers a broad range of architectures and use-cases. We also inherit the user-centric concerns of Klein et al. (2018): The design of MAMMOTH include *transforms*, externalized computation steps providing users with means to include arbitrary preprocessing to better suit their experimental needs; moreover we provide utilities to assist users with designing configuration files for massive and complex experiments semi-automatically. The MAMMOTH toolkit is available publicly under a CC-BY license and warmly welcomes prospective developers and researchers wishing to contribute or request the implementation of specific features.

2 Related work

While there exist many open-source frameworks for training NMT systems that have been used for experiments in modular NMT, such as fairseq (Ott et al., 2019), to the best of our knowledge none of them is specifically targeted at modularity. MAMMOTH is the first open-source toolkit to jointly address the issues of scalability, multilinguality and modularity.

The most relevant point of comparison would be the AdapterHub of Pfeiffer et al. (2020): It extends the transformers library (Wolf et al., 2020; an NLP-centric model sharing platform and training library) so as to enable training of adapter modules for pre-trained state-of-the-art systems. We base our MAMMOTH modular toolkit on the widely used OpenNMT-py (Klein et al., 2018), a customizable library for training NMT and NLG models with a focus on efficiency based on PyTorch: The more thorough documentation and more systematic organization of OpenNMT-py proved a better starting point for MAMMOTH than the fairseq or transformers libraries.

Another motivation behind MAMMOTH is the lack of a standard for the different architectures of existing works on modular systems. Modularity in multilingual NMT has been addressed in a wide range of custom implementations. One common approach is to train language-specific encoders and

decoders. Vázquez et al. (2020) introduce the attention bridge, an intermediate cross-lingual shared layer in between the encoder and decoder. Similarly, Escolano et al. (2021) train language-specific encoders-decoders without sharing any parameters at all. Purason and Tättar (2022) also exploit this method and explore different sharing strategies for the decoder while keeping the decoder language or language-group-specific. Yuan et al. (2023) experiment with massive multilingual MT and propose a plug-and-play approach with detachable modules per language. Other methods investigate the use of language-specific transformer layers (Pires et al., 2023), by keeping some layers source or target language-specific in the encoder. MAMMOTH supports all of the above, and provides a new unifying standard framework for training and testing modular NMT at scale.

3 Toolkit design

We now turn to a description of our toolkit. This section first details the requirements for the toolkit Section 3.1, before moving to the design philosophy we adopted in response to the practical needs outlined in Section 3.2. Finally, we list and describe all major components of our toolkit in Section 3.3.

3.1 Required functionalities

Broad architecture coverage. We aim at a toolkit that covers the major modular architectures proposed in related work (see Section 2) in order to allow systematic studies over a range of implementations in a single all-encompassing framework, ruling out cross-framework variation that prevents a fair comparison of approaches.

In practice, the MAMMOTH toolkit focuses on architectures where modular components can be defined *a priori* and operate as *separable units*. The goal is to allow efficient training of large-scale modular systems from scratch with the possibility of a flexible asymmetric distribution of components across large compute clusters. Furthermore, we aim for efficient inference with a decomposable network where only necessary components need to be loaded for specific tasks. This leaves out sub-network selection approaches, such as the Language Specific Sub-network architecture (LASS; Lin et al., 2021) where a unified multilingual network is split down into language-specific parameters. This also removes dynamic routing approaches, such as the Mixture-of-Experts (MoE;

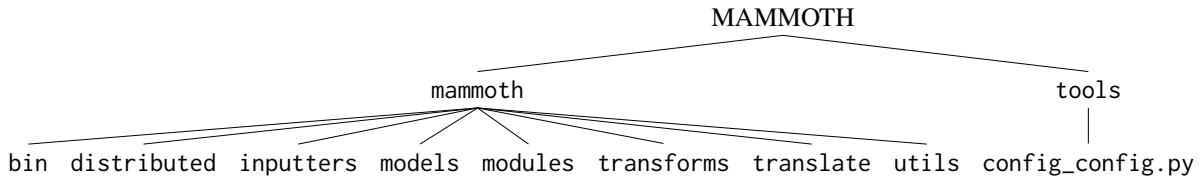


Figure 1: Code repository structure overview

Shazeer et al., 2017) architecture, where a model learns a gating function that activates modules in the network. In such architectures, it is not possible to determine in advance which parameters will be active—and thus all parameters need to be loaded into memory. In contrast, the component-level modularity used in MAMMOTH allows loading only the necessary parameters.

Efficient Training. An important challenge that modular approaches are faced with is owed to their principled sparsity: It is not necessarily feasible—and most often not desirable—to host a copy of all modules on every available computation device. As such, modules have to be assigned to specific devices, which in turn entails that a massively modular system must also be able to handle communication between any two devices that both host a copy of the same module. Minimizing the necessary communication across devices through optimal device assignments of modules becomes a critical aspect for efficient modular training. A practical reality of multilingual and multitask settings is that often data is not equally available for all languages and tasks. Natively handling skewed datasets and ensuring that no computation device is ever idle is also crucial to efficiency.

3.2 Design principles

The MAMMOTH toolkit is designed around the concept of a *task*. A task is the conjunction of three elements, kept constant during the whole of training:

- (i) a set of modules;
- (ii) a set of preprocessing steps; and
- (iii) a single dataset (typically a parallel corpus).

In short, a task corresponds to a specific model behavior. In translation settings, a task will therefore correspond to a specific translation direction (say translating from Swahili to Catalan): All training datapoints for this direction (i) must involve the same modules (pertaining to Swahili encoding and

Catalan decoding); (ii) must be preprocessed with the same tokenizers; and (iii) can be grouped into a single bitext.

We furthermore enforce that a task is tied to a specific compute node and device; in other words, we associate each task with an available GPU, and host a copy of all modules relevant to that task on said GPU. This greatly simplifies questions of device allocation and communication efficiency, since we can examine how allocating specific tasks to specific GPUs will impact communication across devices. In short, we can aim to minimize communication by associating tasks that involve the same components on the same computation device, thereby limiting the number of module copies for which gradient would need to be synchronized.

In Figure 2, we showcase a few examples of configurations file snippets to illustrate how tasks can be defined. In essence, the configuration file expects the `tasks` key to be a mapping of task identifiers (e.g., “`train_bg-en`”) to task definitions. Each task definition must explicitly state the sequence of modules to be used for the encoders and decoders (or “sharing groups”). This allows for a highly flexible modular configuration, ranging from systems that only involve task-specific modules to non-modular systems. The current main limitations we impose are that (i) all tasks must involve the same number of modules,¹ and (ii) module definitions are specific to a given sequential position. In other words, defining two tasks with encoder sharing groups `[x, y]` and `[y, x]` entails defining four modules, not two. These limitations however allow us to factor out the number of layers each module has, by means of the `enc_layers` and

¹Note that this does not entail that all tasks involve the same degree of sharing: By setting up task-specific modular components, one can easily define a pipeline that formally contains the same number of modules (so as to satisfy this requirement) and that is only applied to a given task. To take a concrete example, we could define an NMT system where encoders would be comprised of one language-specific module and one language family-specific module: Isolated languages such as Basque would therefore not share any of their encoder parameters with any other languages.

```

1 tasks:
2   train_bg-en:
3     src_tgt: bg-en
4     enc_sharing_group: [bg]
5     dec_sharing_group: [en]
6     node_gpu: "0:0"
7     path_src: /path/to/train.bg-en.bg
8     path_tgt: /path/to/train.bg-en.en
9   train_cs-en:
10    src_tgt: cs-en
11    enc_sharing_group: [cs]
12    dec_sharing_group: [en]
13    node_gpu: "0:1"
14    path_src: /path/to/train.cs-en.cs
15    path_tgt: /path/to/train.cs-en.en
16  train_en-cs:
17    src_tgt: en-cs
18    enc_sharing_group: [en]
19    dec_sharing_group: [cs]
20    node_gpu: "0:1"
21    path_src: /path/to/train.cs-en.cs
22    path_tgt: /path/to/train.cs-en.en
23
24  enc_layers: [6]
25  dec_layers: [6]

```

```

1 tasks:
2   train_bg-en:
3     src_tgt: bg-en
4     enc_sharing_group: [bg, all]
5     dec_sharing_group: [en]
6     node_gpu: "0:0"
7     path_src: /path/to/train.bg-en.bg
8     path_tgt: /path/to/train.bg-en.en
9   train_cs-en:
10    src_tgt: cs-en
11    enc_sharing_group: [cs, all]
12    dec_sharing_group: [en]
13    node_gpu: "0:1"
14    path_src: /path/to/train.cs-en.cs
15    path_tgt: /path/to/train.cs-en.en
16  train_en-cs:
17    src_tgt: en-cs
18    enc_sharing_group: [en, all]
19    dec_sharing_group: [cs]
20    node_gpu: "0:1"
21    path_src: /path/to/train.cs-en.en
22    path_tgt: /path/to/train.cs-en.en
23
24  enc_layers: [4, 4]
25  dec_layers: [4]

```

```

1 tasks:
2   train_bg-en:
3     src_tgt: all-all
4     enc_sharing_group: [all]
5     dec_sharing_group: [all]
6     node_gpu: "0:0"
7     path_src: /path/to/train.bg-en.bg
8     path_tgt: /path/to/train.bg-en.en
9     transforms: [prefix]
10    src_prefix: "<to_en>"
11  train_cs-en:
12    src_tgt: all-all
13    enc_sharing_group: [all]
14    dec_sharing_group: [all]
15    node_gpu: "0:1"
16    path_src: /path/to/train.cs-en.cs
17    path_tgt: /path/to/train.cs-en.en
18    transforms: [prefix]
19    src_prefix: "<to_en>"
20  train_en-cs:
21    src_tgt: all-all
22    enc_sharing_group: [all]
23    dec_sharing_group: [all]
24    node_gpu: "0:1"
25    path_src: /path/to/train.cs-en.en
26    path_tgt: /path/to/train.cs-en.cs
27    transforms: [prefix]
28    src_prefix: "<to_cs>"
29
30  enc_layers: [6]
31  dec_layers: [6]

```

(a) Example configuration for task-specific encoders and decoders

(b) Example configuration for arbitrarily shared layers in encoders, and task-specific decoders

(c) Example configuration for a non-modular multilingual system. A lack of target language specific parameters necessitates adding a language token using a prefix transform.

Figure 2: Snippets from example configurations

dec_layers keys.

3.3 Implementation

Our code-base is historically based on the OpenNMT-py library (Klein et al., 2018). Although the changes accrued to convert it to a modular framework have proven significant enough to require deep refactoring and restructuring, readers may find referring to Klein et al. (2018) a useful addition to the present description to cover the basic aspects of the NMT toolkit. Some practical implementation choices we inherit from OpenNMT-py include the fact that our framework is based on PyTorch (Paszke et al., 2019) as well as the presence of *transforms* and YAML-based configuration files.

Figure 1 provides an overview of the major elements included in the MAMMOTH toolkit. Utilities listed in the `tools` are intended to facilitate setting up training, conducting experiments, or evaluating models. The core source-code itself is filed under the `mammoth` directory. The source files are grouped in eight python submodules.

The bin submodule. The `bin` submodule contains utilities for training modular systems and translating using MAMMOTH systems.

The transforms submodule. The `transforms` submodule contains a list of default transforms that

toolkit users can easily expand to suit their preprocessing needs. Currently, the MAMMOTH toolkit contains transforms for: external tokenization (e.g., using SentencePiece, Kudo and Richardson, 2018) and subword regularization (Kudo, 2018); denoising auto-encoding task, using a BART-style (Lewis et al., 2020) or a MASS-style (Song et al., 2019) objective; filtering low-quality parallel sentences on the fly; and injecting prefixes, such as control and language tokens or decoder-side prompts.

The distributed submodule. The `distributed` submodule contains the core implementation of the conceptual tasks we outlined in Section 3.2. It also defines routines for efficient communication in modular settings. In particular, we define an algorithm for broadcasting and synchronizing gradients across all tasks shown in Algorithm 1: The gist of it is that we are able to use the a priori structure to omit communication of parameters that are not on the device at all. For the modules on the device, we signal in how many tasks it has been used, and therefore has a gradient, and rely on this information to sum and renormalize gradients appropriately. The `distributed` submodule also contains explicit implementations for computation contexts (distributed vs. single-GPU vs. CPU-bound training), as well as logic to handle uneven task distributions.

Algorithm 1 Gradient accumulation and communication for one datapoint of task \mathcal{T}

Require: set of modules in the complete model,

$$C = \{C_1, \dots, C_u\}$$

Require: set of modules on the device,

$$D = \{D_1, \dots, D_v\} \subseteq C$$

Require: ordered set of modules used in task \mathcal{T} ,

$$T = \{T_1, \dots, T_w\} \subseteq D$$

Require: inputs and labels, $\langle \mathbf{x}, \mathbf{y} \rangle \in \mathcal{D}_{\mathcal{T}}$

▷ forward pass

$$\mathbf{h}_0 \leftarrow \mathbf{x}$$

for $i \in \{1, \dots, w\}$ **do**
 $\mathbf{h}_i \leftarrow T_i(\mathbf{h}_{i-1})$
end for

▷ communicate modules used

$$\text{ready}_T \leftarrow (\mathbf{1}_T(D_1), \dots, \mathbf{1}_T(D_v))$$
$$\mathbf{n} \leftarrow \text{broadcast ready}_T, \text{reduce by sum}$$

▷ backward pass

for $j \in \{1, \dots, v\}$ **do**
 $\Delta D_j \leftarrow \begin{cases} -\nabla_{\mathbf{h}_i} D_j & D_j \in T \\ 0 & D_j \notin T \end{cases}$
 $\Delta D_j \leftarrow \text{broadcast } \Delta D_j, \text{reduce by sum}$
 $\Delta D_j \leftarrow \Delta D_j / \mathbf{n}_j$
end for

The inputters submodule. The inputters submodule contains all code logic for handling data: It provides objects for representing parallel corpora, handling batching through reservoir sampling, and multiplexing batch streams from multiple tasks whenever more than one task is allocated to a given device.

The modules and models submodule. These two python submodules contain code logic for defining specific PyTorch components and grouping them together in coherent models.

The translate submodule. The translate submodule contains all code logic for handling inference.

The utils submodule. The utils submodule regroups all remaining code logic, including optimizers, loss computation, early stopping and tensorboard reporting.

The config-config tool. We prefer explicit over implicit when it comes to the configuration yaml file, even though the configuration can become verbose and repetitive. The config-config tool makes configuring MAMMOTH more user friendly. When given a simple meta-configuration file (see Appendix C for an example), it can perform the following operations to generate the complete configuration:

- Path templating for massively multilingual corpora with various directory structures (See Appendix A for details),
- Find which tasks have data in the corpus,
- Determine task weighting and curriculum,
- Determine language groups by clustering,
- Configure the layerwise parameter sharing groups of tasks (See Appendix B for details),
- Ensure that the correct *transforms* are set for translation and denoising autoencoder tasks,
- Allocate tasks to nodes and GPUs. A local search procedure is used, taking into account parameter sharing groups and tasks delayed by curriculum weighting.
- Determine the adapter configuration for each task.

4 Performances

We utilize the Europarl dataset (Koehn, 2005) - a multilingual resource extracted from European Parliament proceedings containing texts in 21 European languages - for model training to showcase the efficiency of our toolkit. We report performances on the Europarl dataset across various parameter-sharing schemes and computing clusters.

Modeling. We use a SentencePiece model trained on OPUS Tatoeba Challenge data with 64k vocabulary size.² We adopt three sharing schemes: 1) a language-specific one that uses a balanced architecture with a 6-layer transformer encoder and a 6-layer decoder for each language, 2) a partially shared one that has eight layers of encoders (4 shared and language-specific layers) and four layers of decoders, and 3) a fully shared one with 9-layer encoder and 4-layer decoder for all languages. For the transformer network, we enable position encoding and use a dimension of 512 and 8 attention heads. The feed-forward dimension within the transformer is 2048.

Setup. We utilize two types of GPUs: NVIDIA V100 and A100. We run the toolkit with Python 3.9.16 and PyTorch 2.1.0. We use a maximum sequence length for source and target languages

²<https://object.pouta.csc.fi/Tatoeba-MT-spm/opusTC.mul.64k.spm>

of 200, and a batch size of 4096 tokens. The detailed setup guide for the experiment is available in our documentation,³ including data processing, configurations, and launching scripts.

Benchmarking results. We studied the performance of MAMMOTH on CSC’s NVIDIA V100 and A100 clusters, where each node contains four NVIDIA V100/A100 GPUs connected via NVLink and nodes are interconnected via InfiniBand. Scaling benchmarks were undertaken such that the number of tasks increases proportional to the number of allocated GPUs as we are interested in the ability to scale to larger problems with similar load per GPU.

In practice, we train one task on each available GPU for this benchmark. For benchmarking purposes, this task is defined over synthetic data derived from the Europarl dataset (Koehn, 2005): We concatenate bi-texts for all available translation directions, and then randomly split it into 20 sub-corpora. While individual data-points remain coherent, this shuffling process allows us to sidestep concerns about variation in linguistic factors, such as sentence length, that were found to be a concern in an earlier iteration of this benchmark.

Moreover, we studied task scaling with three different approaches. Firstly, an approach where all source languages use language-specific encoders and decoders, denoted by “independent.” Secondly, an approach where all source languages use the same shared encoder but decoders are target language specific, denoted by “shared.” Finally, an approach denoted as “partially shared”, where the encoder begins with a stack of language-specific layers, followed by a stack of shared encoder layers, finally passing information to language-specific decoders, similar to the setup discussed in Figure 2b.

Figure 3 shows task scaling benchmarks from a single node up to 5 nodes using the synthetic dataset. We achieve nearly ideal scaling in all of the scenarios: Drops in performance compared to our ideal reference are limited to 5% at most.

To give more context to the benchmarks, we measured the memory footprint and utilizations independently on each GPU. The mean utilization percentage with the five nodes run with language-specific encoders and decoders was 85%, and the device memory footprint was 7.7G out of 32G. Furthermore, we also did performance profiling

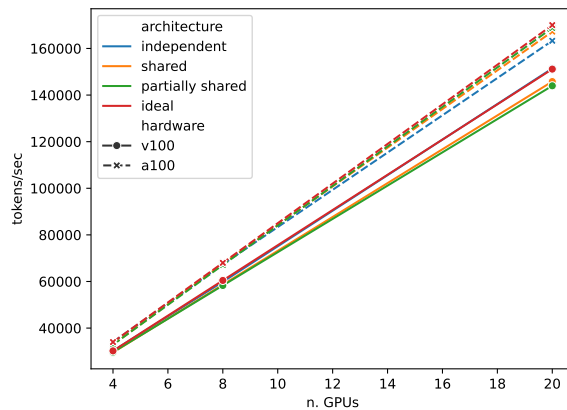


Figure 3: Scaling on V100 and A100 clusters

using NVIDIA Nsight Systems tool to better understand the communication, using the 2-node setup. All communication in the code is outsourced to the NVIDIA Collective Communications Library (NCCL) and at present, it appears that approximately 28 % of the GPU-kernel execution time is spent on NCCL Allreduce. Further communication studies and optimizations with different architectures and setups remain subject to future work. Nevertheless, as we are able to achieve linear scaling beyond a single node, it suggests that the communication overheads are alleviated at scale (relative to tokens processed per second).

Environmental costs. We run the benchmarking experiments on CSC’s carbon-neutral data center powered by hydropower. We measure our carbon footprint using the direct eq. CO₂ of 0kg/kWh and indirect of 0.024kg/kWh as the carbon efficiency.⁴ Each benchmarking experiment runs for around one hour. Our total carbon footprint is 0.11 kg eq. CO₂. The estimated carbon footprint for training a model over 24 hours is 0.14 kg eq. CO₂.

5 Conclusions and next developments

In this paper, we have introduced MAMMOTH, a toolkit for training modular encoder-decoder neural networks at scale. The toolkit is publicly available under a CC-BY license, and we warmly welcome the help of new developers and researchers wishing to extend it.


Further planned development of the MAMMOTH toolkit will specifically focus on the following elements:

³https://helsinki-nlp.github.io/mammoth/examples/train_mammoth_101.html

⁴https://www.syke.fi/fi-FI/Tutkimus_kehittaminen/Kiertotalous/Laskurit/YHiilari

- Interfacing our toolkit with the popular HuggingFace framework, to allow a wider diffusion of MAMMOTH-based modules and reusing existing foundation models for the initialization of modular systems
- Interfacing the MAMMOTH toolkit with the OPUS ecosystem (Tiedemann, 2012), and in particular the OpusFilter tools (Aulamo et al., 2020) so as to delegate data selection to a dedicated third party.
- Providing support for partially frozen modular systems, which would enable adapter-style parameter-efficient fine-tuning.
- Continuing our work of including modular approaches, in particular continuous prefixes.

Acknowledgments

 This work is part of the FoTran project, funded by the European Research Council (ERC) under the EU's Horizon 2020 research and innovation program (agreement № 771113). We also thank the CSC-IT Center for Science Ltd., for computational resources.

References

- Mikko Aulamo, Sami Virpioja, and Jörg Tiedemann. 2020. [OpusFilter: A configurable parallel corpus filtering toolbox](#). In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 150–156, Online. Association for Computational Linguistics.
- Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. [Unsupervised cross-lingual representation learning at scale](#). In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, Online. Association for Computational Linguistics.
- Carlos Escolano, Marta R. Costa-jussà, José A. R. Fonollosa, and Mikel Artetxe. 2021. [Multilingual machine translation: Closing the gap between shared and language-specific encoder-decoders](#). In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 944–948, Online. Association for Computational Linguistics.
- Guillaume Klein, Yoon Kim, Yuntian Deng, Vincent Nguyen, Jean Senellart, and Alexander M. Rush. 2018. [OpenNMT: Neural machine translation toolkit](#).
- Philipp Koehn. 2005. [Europarl: A parallel corpus for statistical machine translation](#). In *Proceedings of Machine Translation Summit X: Papers*, pages 79–86, Phuket, Thailand.
- Taku Kudo. 2018. [Subword regularization: Improving neural network translation models with multiple subword candidates](#). In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia. Association for Computational Linguistics.
- Taku Kudo and John Richardson. 2018. [SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing](#). In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. [BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension](#). In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.
- Zehui Lin, Liwei Wu, Mingxuan Wang, and Lei Li. 2021. [Learning language specific sub-network for multilingual machine translation](#). In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 293–305, 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*. 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 Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. [PyTorch: An imperative style, high-performance deep learning library](#). In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.
- Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2021. [AdapterFusion: Non-destructive task composition for transfer learning](#). In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 487–503, Online. Association for Computational Linguistics.

- Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. [AdapterHub: A framework for adapting transformers](#).
- Telmo Pires, Robin Schmidt, Yi-Hsiu Liao, and Stephan Peitz. 2023. [Learning language-specific layers for multilingual machine translation](#). In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14767–14783, Toronto, Canada. Association for Computational Linguistics.
- Taido Purason and Andre Tättar. 2022. [Multilingual neural machine translation with the right amount of sharing](#). In *Proceedings of the 23rd Annual Conference of the European Association for Machine Translation*, pages 91–100, Ghent, Belgium. European Association for Machine Translation.
- Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.
- Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. [MASS: Masked sequence to sequence pre-training for language generation](#). In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5926–5936. PMLR.
- Jörg Tiedemann. 2012. Parallel data, tools and interfaces in OPUS. In *Lrec*, volume 2012, pages 2214–2218. Citeseer.
- Raúl Vázquez, Alessandro Raganato, Mathias Creutz, and Jörg Tiedemann. 2020. [A systematic study of inner-attention-based sentence representations in multilingual neural machine translation](#). *Computational Linguistics*, 46(2):387–424.
- Zirui Wang, Zachary C. Lipton, and Yulia Tsvetkov. 2020. [On negative interference in multilingual models: Findings and a meta-learning treatment](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4438–4450, Online. Association for Computational Linguistics.
- 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, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. [HuggingFace’s transformers: State-of-the-art natural language processing](#).
- Fei Yuan, Yinquan Lu, Wenhao Zhu, Lingpeng Kong, Lei Li, Yu Qiao, and Jingjing Xu. 2023. [Lego-MT: Learning detachable models for massively multilingual machine translation](#). In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 11518–11533.

A Corpus path templating

Paths to corpora are specified using path templates, which can contain variables that will be substituted by the `config-config` tool.

Directional corpus mode. For corpora where the two translation directions of a language pair are distinguished from each other.

src_lang The source language of the task.

tgt_lang The target language of the task.

lang_pair `src_lang-tgt_lang` for convenience.

Symmetric corpus mode. For corpora where a language pair uses the same files for both translation directions.

lang_a The alphabetically first language.

lang_b The alphabetically second language.

side_a ‘src’ if the language pair is used in the “forward” direction, otherwise ‘trg’. Note that the abbreviation for target is ‘trg’, not ‘tgt’.

side_b ‘trg’ if the language pair is used in the “forward” direction, otherwise ‘src’.

sorted_pair the source and target languages in alphabetical order, separated by a hyphen.

As an example, let’s say that the corpus contains two files `eng-ben/train.src.gz` (English side) and `eng-ben/train.trg.gz` (Bengali side). The data should be used symmetrically for both Bengali-to-English and English-to-Bengali directions. For the first, `lang_pair` and `sorted_pair` are the same. For the second, `lang_pair` is “eng-ben”, but `sorted_pair` is “ben-eng”.

Thus, in order to use the files in the correct order, you should use the source path template `{sorted_pair}/train.{side_a}.gz`, and `{sorted_pair}/train.{side_b}.gz` as the target path template.

B Layerwise parameter sharing

The parameter sharing architecture is defined as two concatenations of modules, one for the encoder and one for the decoder. Each module has a specified number of layers, and a parameter sharing pattern. The following parameter sharing patterns are available in `config-config`. Note that arbitrary sharing patterns are possible when generating the configuration file by other means.

FULL fully shared parameters. Will be named using the constant “full”.

SRC_GROUP groupwise shared parameters. Will be named according to the cluster id of the source language.

TGT_GROUP groupwise shared parameters. Will be named according to the cluster id of the target language.

GROUP groupwise shared parameters. Same as **SRC_GROUP** for encoder and **TGT_GROUP** for decoder. For convenience.

SRC_LANGUAGE language specific parameters. Will be named according to the source language code.

TGT_LANGUAGE language specific parameters. Will be named according to the target language code.

LANGUAGE language specific parameters. Same as **SRC_LANGUAGE** for encoder and **TGT_LANGUAGE** for decoder. For convenience.

Note that it is possible to have target-language-dependent modules in the encoder, by using **TGT_LANGUAGE** or **TGT_GROUP** in the definition of the encoder sharing patterns⁵.

C Example meta-config

We include a practical example of a meta-configuration file in Figure 4.

Note that the number of GPU devices per node (`n_gpus_per_node`) and the maximum number of tasks to allocate per GPU (`n_slots_per_gpu`) should be configured according to the cluster hardware.

⁵Source-language dependent modules in the decoder are possible as well.

```

1 config_config:
2   src_path: "/data/Tatoeba-Challenge/{sorted_pair}/train.src.gz"
3   tgt_path: "/data/Tatoeba-Challenge/{sorted_pair}/train.trg.gz"
4   ae_path: "/data/Tatoeba-Challenge/monolingual_filtered/{tgt_lang}.txt.gz"
5   valid_src_path: "/data/Tatoeba-Challenge/dev/dev.{sorted_pair}.{src_lang}"
6   valid_tgt_path: "/data/Tatoeba-Challenge/dev/dev.{sorted_pair}.{tgt_lang}"
7   autoencoder: True
8   n_groups: 1
9   keep_lc_cache: True
10  use_weight: True
11  use_introduce_at_training_step: False
12  split_large_language_pairs: 0.25
13  temperature: 0.2
14  ae_weight: 0.1
15  zero_shot: True
16  transforms:
17    - filtertoolong
18    - sentencepiece
19    - prefix
20    - filtertoolong
21  ae_transforms:
22    - filtertoolong
23    - sentencepiece
24    - denoising
25    - prefix
26    - filtertoolong
27  enc_sharing_groups:
28    - FULL
29    - SRC_LANGUAGE
30    - FULL
31    - TGT_LANGUAGE
32    - FULL
33  dec_sharing_groups:
34    - TGT_LANGUAGE
35  translation_config_dir: config/translation/tatoeba.pires23
36  n_gpus_per_node: 8
37  n_slots_per_gpu: 6
38  groups:
39    afr: "all"
40    asm: "all"
41    # omitted for brevity: more languages
42    vie: "all"
43    zho: "all"
44
45  src_vocab:
46    afr: "/sentencepiece/opusTC.afr.32k.spm.vocab"
47    asm: "/sentencepiece/opusTC.asm.32k.spm.vocab"
48    # omitted for brevity: more languages
49    vie: "/sentencepiece/opusTC.vie.32k.spm.vocab"
50    zho: "/sentencepiece/opusTC.zho.32k.spm.vocab"
51  tgt_vocab:
52    # omitted for brevity: same as src_vocab
53  overwrite: False
54
55  src_subword_model: "/sentencepiece/opusTC.{src_lang}.32k.spm"
56  tgt_subword_model: "/sentencepiece/opusTC.{tgt_lang}.32k.spm"
57  enc_layers: [2, 2, 8, 3, 1]
58  dec_layers: [3]
59
60  ##### Tuned in HPO
61  learning_rate: 1e-4
62  adam_beta1: 0.9
63  adam_beta2: 0.98
64  dropout: 0.01
65  weight_decay: 0.001
66  label_smoothing: 0.1
67  accum_count: 16
68
69  ##### Constant
70  model_dim: 512
71  transformer_ff: 2048
72  heads: 8
73  warmup_steps: 10000
74  decay_method: linear_warmup
75  train_steps: 150000
76  valid_steps: 5000
77  save_checkpoint_steps: 5000
78  optim: adamw
79  max_grad_norm: 5.0
80  batch_size: 14000
81  batch_type: tokens
82  normalization: tokens
83  valid_batch_size: 256
84  max_generator_batches: 2
85  bridge: false
86  encoder_type: transformer
87  decoder_type: transformer
88  param_init: 0.0
89  param_init_glorot: true
90  position_encoding: true
91  report_every: 100
92  keep_checkpoint: 3
93  seed: 3435
94  model_type: text
95
96  ##### Sentencepiece
97  src_subword_nbest: 5
98  tgt_subword_nbest: 5
99
100 ##### Filter
101 src_seq_length: 200
102 tgt_seq_length: 200
103
104 ##### Bart
105 denoising_objective: bart
106 mask_length: span-poisson
107 poisson_lambda: 3.5
108 mask_ratio: 0.5
109 random_ratio: 0.1
110 replace_length: 1
111
112 structured_log_file: "logs/tatoeba.pires23.jsonl"
113 save_model: models/tatoeba.pires23

```

Figure 4: Example meta-config