DMS: A System for Delivering Dynamic Multitask NLP Tools

Haukur Páll Jónsson[1][a] and Hrafn Loftsson[2][b]

[1]*Miðeind ehf., Reykjavík, Iceland*

[2]*Department of Computer Science, Reykjavik University, Iceland*

Keywords: NLP, Multitask, Icelandic, PoS, Lemmatization.

Abstract: Most NLP frameworks focus on state-of-the-art models which solve a single task. As an alternative to these frameworks, we present the Dynamic Multitask System (DMS), based on native PyTorch. The DMS has a simple interface, can be combined with other frameworks, is easily extendable, and bundles model downloading with an API and a terminal client for end-users. The DMS is flexible towards different tasks and enables quick experimentation with different architectures and hyperparameters. Components of the system are split into two categories with their respective interfaces: encoders and decoders. The DMS targets researchers and practitioners who want to develop state-of-the-art multitask NLP tools and easily supply them to end-users. In this paper, we, first, describe the core components of the DMS and how it can be used to deliver a trained system. Second, we demonstrate how we used the DMS for developing a state-of-the-art PoS tagger and a lemmatizer for Icelandic.

1 INTRODUCTION

The development of state-of-the-art NLP tools has become easier in recent years, partly due to the emergence of quality frameworks, implemented in a single, easy to use, language. For example, FLAIR (Akbik et al., 2019), Transformers (Wolf et al., 2020), AllenNLP (Gardner et al., 2018), fastai (Howard and Gugger, 2020), and fairseq (Ott et al., 2019)) are all relatively new frameworks, which are implemented in Python and have a backbone written in a faster, compiled language.

Most NLP frameworks, like the previously-mentioned, focus on solving a single task. Furthermore, to make the developer experience more streamlined, they often provide a plethora of abstractions, which the developer is expected to use, but can cause a steep learning curve.

As an alternative to these frameworks, we present a system called Dynamic Multitask System (DMS), which focuses on combining multiple tasks into a single model – a multitask model. The DMS, which is based on native PyTorch, has a simple interface, can be combined with other frameworks, is easily extendable, and bundles model downloading with an API and a terminal client for end-users.

The DMS targets researchers and practitioners who want to develop state-of-the-art NLP tools and easily supply them to end-users. The system's flexibility towards different tasks and its simple interface enables quick experimentation with different architectures and hyperparameters. The current implementation focuses on Part-of-Speech (PoS) tagging and lemmatization, but can easily be extended to other tasks, e.g. sentence classification or open text generation. The code is implemented in Python 3.8/PyTorch 1.8 and is published with the Apache 2.0 license[1].

The DMS is designed from the ground up to be a dynamic multitask system. For example, the system can be used to train a model which can produce PoS tags and/or lemmas without having to duplicate parts of the code. To achieve this, we split components of the system into two categories: encoders and decoders, with their respective interfaces. The system then relies on these components to do all the necessary pre- and post-processing.

Let us contrast the dynamic multitask approach, proposed in this paper, to a static multitask approach, i.e. an approach which solves a specific multitask problem by making hard architectural assumptions. The dynamic approach allows for easier architecture experimentation because the components are not as tightly coupled. If components are tightly coupled

[a] https://orcid.org/0000-0001-9615-3455

[b] https://orcid.org/0000-0002-9298-4830

[1]https://github.com/cadia-lvl/POS

Jónsson, H. and Loftsson, H.
DMS: A System for Delivering Dynamic Multitask NLP Tools.
DOI: 10.5220/0011005000003116
In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022) - Volume 1*, pages 504-510
ISBN: 978-989-758-547-0; ISSN: 2184-433X

and one wants to carry out an ablation study of the component's impact on the overall performance, one needs to adjust the code in multiple locations in the training pipeline: in the preprocessing step, in dependant components, in the loss function, and when mapping the model's output to human-readable strings. Instead, we suggest a simple interface and a reference implementation of multiple components which addresses these problems and allows for quick experimental iterations. The most notable trade-off using this approach is computational speed during training, as we tie the preprocessing step into the training loop. The DMS is therefore not suitable for training models which rely on large amounts of data (over a few GBs), but rather for less data-intensive tasks. We believe that this is an acceptable trade-off for the suggested use case.

Furthermore, the previous paragraph only addresses the problems a researcher/practitioner needs to be aware of, but not how the trained model will be consumed by the end-user. The end-user wants to be able to use a trained model, with as little effort as possible. To achieve this, we use off-the-shelf solutions for loading code and trained models for the end-user along with an API and a terminal client which leverage the dynamic design of the system.

The DMS system should not be considered as a *framework* as it does not try to push many abstractions onto the developer. It rather uses PyTorch primitives and can be used in conjunction with other existing *text embedding* frameworks (Huggingface Transformers, FLAIR, etc.) and can be made to fit other PyTorch *training* frameworks. The system should be easily adoptable by other researchers/practitioners working on state-of-the-art NLP tools who, additionally, want to release those tools to end-users in an easy to use manner.

Originally, our goal was to develop a PoS tagger and a lemmatizer for Icelandic as a part of the Language Technology Programme for Icelandic 2019-2023 (Nikulásdóttir et al., 2020). The programme combines software development and research, i.e. the tools need to be developed and delivered to end-users. In order to deliver a joint high-performing PoS tagger and a lemmatizer, we needed to experiment with combinations of multiple components. None of the existing frameworks had an off-the-shelf solution for this problem – they make solving certain problems easy but at the cost of a lack of flexibility. Thus, we needed to develop our own system which could leverage model implementation available in state-of-the-art frameworks. Our resulting PoS tagger for Icelandic is state-of-the-art, achieving an accuracy of 97.84%.

The rest of the paper is structured as follows: In Section 2, we present the DMS system. In Section 3, we present our implementation and evaluation results for PoS tagging and lemmatization for Icelandic. Finally, we conclude in Section 4.

2 THE DYNAMIC MULTITASK SYSTEM

In this section, we describe the core components of the DMS, namely the *Encoder* and the *Decoder*. We describe the interface and how it is used during training and inference. We then list the currently implemented components and explain how a trained system is delivered with an API and a terminal client to the end-user.

2.1 The Core

The core part of the system mainly consists of two interfaces and a class which consists of a sequence of implementations of these interfaces. The two interfaces are *Encoder* and *Decoder*. The module which combines the encoders and decoders is aptly named *EncodersDecoders*. All of them are PyTorch *Modules*. To implement a PyTorch Module one needs to implement the *forward* method, which is called for each forward step of the network. An overview of the system can be seen in Figure 1.

The Encoder takes care of preprocessing a batch of input sequences and encodes them for downstream modules. An Encoder is a PyTorch *Module* which implements the *BatchPreprocess* interface and has an *output_dim* property. The *BatchPreprocess* interface defines a function which accepts a batch of inputs and preprocesses them. Thus, an implementation of an Encoder defines how the input sequence should be transformed from the text sequence to an *encoding* via the *preprocess* and *forward* steps.

Similarly, the Decoder takes care of ingesting the encodings and postprocessing them to the expected output. A Decoder is a PyTorch *Module*, which implements the *BatchPostprocess* interface, a method called *add_targets* and has two properties: *weight* and *output_dim*. The *BatchPostprocess* interface defines a function which accepts batch of inputs which have been passed through the *forward* method of the Decoder and maps it to a sequence of strings. During training, the *add_targets* method takes care of mapping the target output to a format expected by the decoder's loss function. When computing the total loss, the decoder's loss is weighted by the defined *weight*.

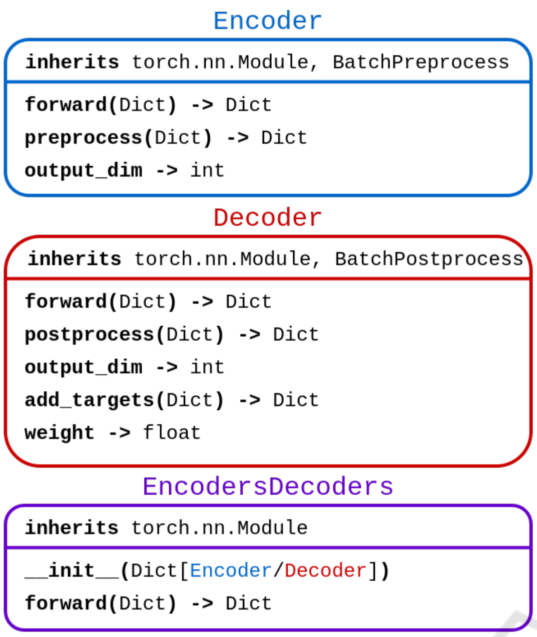

Figure 1: An overview of the DMS interface. The Encoder transforms a text sequence to an encoding via preprocess and forward. The Decoder transforms encodings to a text sequence and further informs the training process.

The *EncodersDecoders* module accepts multiple encoders and decoders and first runs the *preprocess* and *forward* steps for each encoder and then the *forward* step for each decoder. This loop is executed during training and inference and returns the raw predictions for each decoder. The *add_targets* is only called during training as it is only used to compute the loss function. The *BatchPostprocess* is only called during inference and validation.

The main benefits of this approach are:

- **Multiple and Separate Data Processing:** Different encoders can have diverse requirements on their input preprocessing, and the same goes for decoder postprocessing. By making these data processing steps a part of their implementation, a separate, component-aware data-processing pipeline is not required.
- **Dynamic Batch:** By storing all previously computed values of a batch (forward/preprocessing/postprocessing) in a Python dictionary, later components can easily access those values. Indeed, this makes the components order-dependant, but that simply reflects the architecture of the overall system.
- **Experimentation and Ablation:** As all components of the model are dynamically added to create an overall architecture, components of the system can be easily adjusted and ablated.

2.2 Implemented Encoders

For our PoS tagging and lemmatization experiments the model's input is a tokenized sentence. We have implemented several text encoders:

- **CharacterEncoder:** preprocesses the tokens into a sequence of characters indices and then encodes them using a PyTorch *Embedding*.
- **WordEncoder:** preprocesses the tokens to indices which are derived from the training data and then encodes them using a PyTorch *Embedding*.
- **PretrainedWordEncoder:** works the same way as the WordEncoder except the indices and weights are from external sources.
- **CharactersAsTokenEncoder:** a bidirectional RNN (GRU (Cho et al., 2014)) which does no preprocessing, but rather accepts the CharacterEncoder output as input, feeds it to the RNN and returns the last hidden state as well as the output for each timestep.
- **TransformerEncoder:** a BERT-like model (currently, ELECTRA (Clark et al., 2020)) along with the pretrained subword tokenizer (Wu et al., 2016). During preprocessing the tokens are converted to subwords and a *token_start* mask is computed. The subwords are then encoded using the BERT-like model and the last hidden state masked with the *token_start* is returned.
- **SentenceEncoder:** a bidirectional LSTM (Hochreiter and Schmidhuber, 1997) which has no preprocessing step, but rather accepts a list of encodings, which have the same sequence length, concatenates them along the feature dimension and feeds the sequence to the LSTM and returns the output for each timestep.

2.3 Implemented Decoders

We have implemented the following decoders which map the encoded text to the output of the desired task:

- **Tagger:** a sequence tagger used to predict PoS tags. It is a re-implementation of a *classification head* in Huggingface Transformers, i.e. a *dense* layer, followed by a layer *normalization* (Ba et al., 2016), a *relu* activation, and, finally, a *linear* layer with an output dimension equal to the number of classes.
- **Lemmatizer:** an autoregressive character decoder. It is an RNN (GRU) and produces the lemma of a given word, one character at a time.

The input for each time-step is the previous predicted character, a context vector and multiplicative attention vector over the time-sequence of a CharactersAsTokenEncoder (Luong et al., 2015).

- **Structured Tagger:** a multilabel-multiclass sequence tagger. It consists of a Tagger per label, where each label is a sub-category of a PoS tag.

As previously mentioned, each decoder implements two methods; *add_targets* in which the target outputs are mapped to a format suitable for the loss function and *BatchPostprocess* in which the predictions of the decoder are mapped to a sequence of strings.

For each decoder there is an associated loss function which is scaled by the decoder's weight. All the losses are then summed up and a backward step applied to the combined loss.

2.4 Delivering Trained Systems

Delivering an easy-to-use trained system can often be a time-consuming task. The DMS makes this task simpler.

After training, the trained model's weights are stored to disk along with all necessary files required to successfully load the model: the global configuration of the model components and the configuration for each component (e.g. string-to-index mappings, subword tokenizer, etc.). For a model release, these files are packaged and uploaded to a web storage, for example, CLARIN (Hinrichs and Krauwer, 2014).

An API is then defined which handles the interface towards a trained model. The API initializes the model parameters and loads the weights and other necessary files. It then provides easy to use functions based on the defined decoders. This API is then exposed to the end-user via a PyTorch Hub configuration file. The PyTorch Hub configuration also handles model downloading and extraction. The terminal client replicates the functionality of the PyTorch Hub configuration.

3 EXAMPLE IMPLEMENTATION AND RESULTS

In this section, we describe our implementation of PoS tagging and lemmatization for Icelandic. In particular, we go through the development process and experimentation which demonstrates the usefulness of the DMS. At multiple stages in the development process, we released the trained models to end-users. The model's accuracies are summarized in Table 1.

3.1 Reimplementing ABLTagger

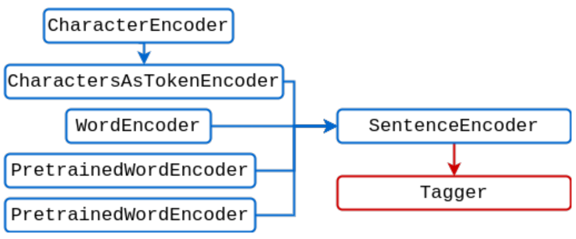

Figure 2: An overview of the improved ABLTagger. It uses the CharacterEncoder, CharactersAsTokenEncoder, WordEncoder, and two different PretrainedWordEncoder. These are then combined using the SentenceEncoder and fed to the Tagger.

We started by reimplementing the PoS tagger (ABLTagger) presented in (Steingrímsson et al., 2019) in PyTorch. It roughly consists of *CharactersAsTokenEncoder*, *WordEncoder*, a *PretrainedWordEncoder* with hand-constructed n-hot vectors based on the the Database of Icelandic Morphology (DIM) (Bjarnadóttir et al., 2019). All of these encoders are then combined using the *SentenceEncoder* and decoded using the sequence *Tagger*. The ABLTagger achieves an accuracy of 95.15% on MIM-GOLD (Loftsson et al., 2010), the standard PoS benchmark for Icelandic.

The input to the model consists of tokenized text which is then further broken down into characters for the *CharactersAsTokenEncoder*. We further require two different token-to-index mappings, one for the vanilla *WordEncoder* and another for the *PretrainedWordEncoder*, as their vocabularies differ.

We performed multiple ablation studies on the individual components to determine the effect of each component. Whilst doing that, we discovered that certain components were under-performing and found better hyper parameters. We also incorporated another *PretrainedWordEncoder* based on fastText (Bojanowski et al., 2017). Here, we found the dynamic nature of the DMS to be helpful in testing different architecture variations.

This improved version of ABLTagger resulted in an increase of accuracy to 95.59%. An overview of the architecture can be seen in Figure 2.

3.2 Incorporating a TransformerEncoder

In the next step, we incorporated a *TransformerEncoder*, an ELECTRA-small model trained on the Icelandic Gigaword Corpus (Steingrímsson et al., 2018). We evaluated multiple configurations of the previous components in conjunction with the TransformerEn-

Table 1: A summary of PoS tagging and Lemmatization experiments performed using the DMS. The results are based on 9-fold cross-validation on MIM-GOLD excluding the "e" and "x" tags. The lemmatization accuracies are based on a non-standard split.

	Accuracy	
System	PoS tagging	Lemmatization
ABLTagger	95.15%	-
Improved ABLTagger	95.59%	-
ELECTRA-small	96.65%	97.54%
ELECTRA-small + DIM	96.65%	98.90%
ELECTRA-base	97.84%	-

coder and the final model achieves an accuracy of 96.65%. Incorporating a TransformerEncoder further increased the complexity of the preprocessing as now we needed to apply a subword tokenizer on the input whilst ensuring that the subword sequence length did not exceed the positional encoding limit on the TransformerEncoder. Furthermore, we needed to ensure that the number of outputs from the TransformerEncoder equalled the number of tokens from the other modules. Here, we found the *BatchPreprocess* interface in the *Encoder* to be helpful.

The ELECTRA-small model was then switched out for an ELECTRA-base model. In the experiments with the ELECTRA-base model, we found that the other components, the *CharactersAsTokenEncoder*, the *WordEncoder* and both the *PretrainedWordEncoders* did not improve the PoS tagging accuracy, resulting in a maximum, state-of-the-art, accuracy of 97.84%.

3.3 Adding the Lemmatizer

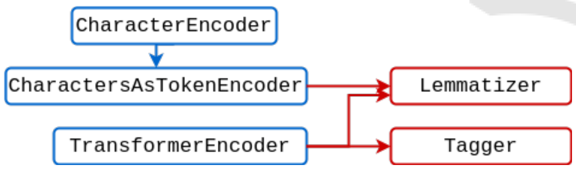

Figure 3: An overview of the joint tagger and lemmatizer. It uses the CharacterEncoder, CharactersAsTokenEncoder and the TransformerEncoder. The CharactersAsTokenEncoder is fed to Lemmatizer. The Tagger and Lemmatizer share the TransformerEncoder.

Once the PoS tagging experiments were finished, we trained a stand-alone *Lemmatizer* and a joint *Lemmatizer* and *Tagger*. The joint model can be seen in Figure 3. Both the stand-alone and joint models used a *TransformerEncoder* and a *CharactersAsTokenEncoder*. By comparing the stand-alone model with the joint model, we found that the Lemmatizer in the joint model was under-performing and that the Lemmatizer was negatively affecting the PoS tagger. To attempt to remedy this, we scaled down the loss weight on the Lemmatizer and pretrained the Lemmatizer on data from the DIM, i.e. lemmatization with PoS context, but no sentence context. The model was then fine-tuned on MIM-GOLD.[2] Here, we found the *Decoder* interface of the DMS to be very helpful. We are still not satisfied with PoS tagging performance of the joint model, as the PoS tagging accuracy is still negatively affected by the *Lemmatizer*.

We have yet to experiment with the *Structured Tagger*, in which we predict PoS tag sub-categories, allowing us to predict tags not seen in the training data.[3] We also want to experiment with different approaches for the joint model.

4 CONCLUSIONS

We have presented DMS, the Dynamic Multitask System, and demonstrated its usefulness and simplicity by applying it to PoS tagging and lemmatization for Icelandic. Our PoS tagger achieves state-of-the-art accuracy of 97.84%. Multitask systems are inherently more complex to develop than single-task systems, but the DMS can reduce the development effort for multitask systems. The DMS can be easily extended to different tasks, leverage state-of-the-art text encoders and simplify frequent deliveries to end-users.

We plan to continue developing the DMS, mainly to make it easier to use for the developer. In short, the DMS pushes a lot of the complexity to the system's run configuration. This configuration can become unwieldy, but this can be mitigated by run configuration tools, such as Hydra (Yadan, 2019). Hydra enables the developer to "dynamically create a hierarchical configuration by composition and override it through config files and the command line".

[2]Note that the MIM-GOLD lemma data had not been released at this stage, so we were using a non-standard split.

[3]There are roughly 600 PoS tags in the Icelandic tag set, whereas only about 570 are seen in the training data. The tags contain a structure which we expect the model to be able to learn.

Furthermore, some boiler-plate code required for the training loop could also be reduced with a training framework, such as PyTorch Lightning. PyTorch Lightning is a *lightweight* PyTorch wrapper which reduces the engineering effort required to train models. It reduces the boiler-plate code required to train models on multiple GPUs, different hardware, different floating-point precision etc.

ACKNOWLEDGEMENTS

This project was funded by the Language Technology Programme for Icelandic 2019–2023. The programme, which is managed and coordinated by Almannarómur[4], is funded by the Icelandic Ministry of Education, Science and Culture.

We would like to thank Jón Friðrik Daðason at Reykjavik University for supplying us with the ELECTRA models used in this research.

REFERENCES

Akbik, A., Bergmann, T., Blythe, D., Rasul, K., Schweter, S., and Vollgraf, R. (2019). FLAIR: An easy-to-use framework for state-of-the-art NLP. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, Minneapolis, Minnesota. Association for Computational Linguistics.

Ba, J., Kiros, J., and Hinton, G. E. (2016). Layer normalization. *ArXiv*, abs/1607.06450.

Bjarnadóttir, K., Hlynsdóttir, K. I., and Steingrímsson, S. (2019). DIM: The database of Icelandic morphology. In *Proceedings of the 22nd Nordic Conference on Computational Linguistics*, pages 146–154, Turku, Finland. Linköping University Electronic Press.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. (2020). ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*.

Gardner, M., Grus, J., Neumann, M., Tafjord, O., Dasigi, P., Liu, N. F., Peters, M., Schmitz, M., and Zettlemoyer, L. (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.

Hinrichs, E. and Krauwer, S. (2014). The CLARIN research infrastructure: Resources and tools for eHumanities scholars. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 1525–1531, Reykjavik, Iceland. European Language Resources Association (ELRA).

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

Howard, J. and Gugger, S. (2020). Fastai: A Layered API for Deep Learning. *Information*, 11(2).

Loftsson, H., Yngvason, J. H., Helgadóttir, S., and Rögnvaldsson, E. (2010). Developing a PoS-tagged corpus using existing tools. In *Proceedings of "Creation and use of basic lexical resources for less-resourced languages", workshop at the 7th International Conference on Language Resources and Evaluation (LREC 2010)*, Valetta, Malta.

Luong, T., Pham, H., and Manning, C. D. (2015). Effective Approaches to Attention-based Neural Machine Translation. *ArXiv*, abs/1508.04025.

Nikulásdóttir, A., Guðnason, J., Ingason, A. K., Loftsson, H., Rögnvaldsson, E., Sigurðsson, E. F., and Steingrímsson, S. (2020). Language technology programme for Icelandic 2019-2023. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 3414–3422, Marseille, France. European Language Resources Association.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. (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.

Steingrímsson, S., Helgadóttir, S., Rögnvaldsson, E., Barkarson, S., and Guðnason, J. (2018). Risamálheild: A very large Icelandic text corpus. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Steingrímsson, S., Kárason, Ö., and Loftsson, H. (2019). Augmenting a BiLSTM tagger with a morphological lexicon and a lexical category identification step. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019)*, pages 1161–1168, Varna, Bulgaria. INCOMA Ltd.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S.,

[4]https://almannaromur.is/

Drame, M., Lhoest, Q., and Rush, A. (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.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *ArXiv*, abs/1609.08144.

Yadan, O. (2019). Hydra - A framework for elegantly configuring complex applications. Github.