Reimplementation of (Bradbury et al., 2016): Quasi-Recurrent Neural networks

Leander Girrbach
girrbach@cl.uni-heidelberg.de

Abstract

This report describes my efforts to reimplement the Quasi-Recurrent Neural Network architecture described in Bradbury et al. (2016) and my efforts to replicate their experimental results. Code is provided here.

1 Introduction

The authors of Bradbury et al. (2016) define a neural network architecture for sequence processing which they call "Quasi-Recurrent Neural Network" (QRNN).

In the following, I will briefly state the definition of QRNNs and repeat some extensions of the QRNN as propsed in the paper.

A single QRNN layer is composed of two computation steps:

1. Convolution step
2. Pooling step

Like conventional LSTMs, QRNN layers compute hidden and context states.

1.1 Convolution step

Let $\mathbf{X}$ be the input sequence to the QRNN layer and $\mathbf{W}_z, \mathbf{W}_f, \mathbf{W}_o, \mathbf{W}_i$ weight matrices.

The convolution steps consists of applying 1d-convolution (along the time dimension). I will use hidden units synonymous with filters, because 1 filter amounts to 1 dimension of the convolved output. In the paper this is: $Z = \tanh(\mathbf{W}_z * \mathbf{X})$ where $*$ is the convolution operator.

Depending on the pooling method, up to 3 gates are calculated by applying $1d$-convolution:
$F = \sigma(\mathbf{W}_f * \mathbf{X})$, $O = \sigma(\mathbf{W}_o * \mathbf{X})$,
$I = \sigma(\mathbf{W}_i * \mathbf{X})$.

1.2 Pooling step

The pooling step asserts the possibility of modelling long-range dependencies. It consists of recurrently calculating gated hidden and context states. The authors define 3 such recurrent pooling functions.

f-pooling

$$h_t = f_t \odot h_{t-1} + (1 - f_t) \odot z_t$$

(no context state)

$\odot$ is elementwise multiplication. t denotes the time step.

fo-pooling

$$c_t = f_t \odot c_{t-1} + (1 - f_t) \odot z_t$$
$$h_t = o_t \odot c_t$$

h is for **h**idden state, c is for **c**ontext state.

ifo-pooling

$$c_t = f_t \odot c_{t-1} + i_t \odot z_t$$
$$h_t = o_t \odot c_t$$

1.3 Extensions (from the paper)

Dense connections This means concatenating the input of a QRNN layer to the QRNN layer's output (along the feature dimension). The concatenation then serves as input to the next QRNN layer.

Masked convolutions This extension means prepending padding of $k - 1$ timesteps at the beginning of a sequence. Therefore, each convolution result only depends on information from the same or previous timesteps. The model cannot use information from the future. This is important e.g. for language models.

Also, it ensures that the output sequence has the same length as the input sequence.

Zoneout Zoneout is a dropout-like method to use with the recurrent pooling functions. The authors define zoneout by redefining F

$$F = 1 - \text{dropout}\left(1 - \sigma\left(\mathbf{W}_f * \mathbf{X}\right)\right)$$

2 Methods

All subsequently described implementations are realised in PyTorch (Paszke et al., 2019) (version 1.4.0) and integrated into JoeyNMT (Kreutzer et al., 2019) and Allennlp (Gardner et al., 2017), subsequently (re)using their code as well.

A reference implementation of the QRNN in PyTorch is provided by one of the authors in `https://github.com/salesforce/pytorch-qrnn/`. Besides being only compatible to PyTorch 0.4, the reference has some shortcomings, among which are:

1. No bidirectional mode
2. Kernel width only 1 or 2
3. No fo-pooling or ifo-pooling
4. No implementation of the QRNN Decoder variant

For my reimplementation, I tried to add such missing features. The most important part which I reused from the reference is the CUDA-Kernel for the f-pooling operation. Still, I updated it to exactly match the formulation in the paper[1], also I added a CUDA-Kernel for ifo-pooling (based on the original kernel for f-pooling).

2.1 QRNN as RNN replacement

The implementation in `qrnn.py` includes 2 classes. They are called `QRNN` and `QRNNLayer`.

QRNN The `QRNN` class holds instances of `QRNNLayer` objects and passes the repective input sequence to each QRNN layer.

`QRNN` is designed to implement the API of the PyTorch RNN implementation. The outputs and their semantics are identical. However, it is possible to pass a hidden state to the PyTorch RNN. For compatibility purposes, this is also possible for the `QRNN`, but the passed tensor will be ignored. As suggested in the paper, hidden states will always be initialised with zeros.

Small parts of the code are taken exactly from the PyTorch RNN implementation, especially the naming and storing of layers.

The reason for implementing the PyTorch RNN API is that this enables the model to be easily inserted into frameworks which normally use or expect a RNN, for example JoeyNMT's `RecurrentEncoder` and Allennlp's `PytorchSeq2SeqWrapper`.

QRNNLayer The `QRNNLayer` implements a single layer's convolution step and pooling operations. Convolution uses torch.nn.Conv1d. The recurrent pooling uses an adaptation of the CUDA-kernel from the original Salesforce implementation.

`QRNNLayer` also manages the prepending of padding for masked convolutions. Prepending padding to the sequence is hardcoded, so it cannot be switched off for the machine translation task, which is suggested in the paper.

2.2 QRNN Decoder

The QRNN decoder is implemented by 3 classes: `QRNNDecoder`, `QRNNDecoderLayer`, and `AttentionLayer`.

`QRNNDecoder` is an adaptation (and implemented as subclass) of JoeyNMT's `RecurrentDecoder`. Some parts of the original code are used unchanged.

The main difference to JoeyNMT's `RecurrentDecoder` is that no custom attention module can be used. Also the initialisation of hidden states is changed. Because the convolution step needs access to previous input timesteps, `QRNNDecoder` caches the whole input and hidden sequences.

The main difference to `QRNN` as RNN replacement is that every timestep is calculated individually. This is forced by the `_forward_step`-method of JoeyNMT's `RecurrentDecoder`. Also, in this variant of the encoder-decoder scheme, decoder layers receive a projection of the last hidden state of the respective encoder's QRNN layer as input. This requires the same number of layers in encoder and decoder. Next, I implemented the attention mechanism described in the paper. It is

[1] The reference implementation is equivalent, but uses the f-gate differently. I decided to change this for improved congruency of paper and code.

implemented as a `QRNNDecoderLayer` replacement which only supports fo-pooling. Note that the formulation of attention in the paper requires the same number of hidden units the last layer of encoder and decoder. If the encoder is bidirectional, the last encoder layer has to to have half the number of hidden units of the last decoder layer.

Some of the requirements of the QRNN decoder (mainly access to hidden states of all encoder layers and having to keep hidden states for multiple timesteps during beam search due to the convolution operation) force changes to other parts of JoeyNMT. Therefore, my variant of JoeyNMT currently only supports QRNNs and not any other architecture such as conventional RNN variants or Transformer.

3 Results and Analysis

I try to replicate the experiments as described in the paper. For review classification and language modelling, I use Allennlp (Gardner et al., 2017) as framework, and for character-based machine translation, I use JoeyNMT (Kreutzer et al., 2019) as framework.

3.1 Review Classification

Dataset This task is a binary sentiment classification task using the annotated part of the dataset from (Maas et al., 2011)[2]. It consists of 50000 movie reviews labeled either "positive" or "negative". Labels are balanced, so that 25000 reviews are positive and 25000 are negative. 50% of the data are training data and 50% of the data are test data.

Other than the authors, I do not use an held-out validation set.

As suggested in the paper, my implementation allows for initialising the embeddings with GLoVe-embeddings (Pennington et al., 2014)[3]. I provide a script to download and prepare the embeddings. This makes use of gensim (Řehůřek and Sojka, 2010).

Model used & Hyperparameters I use a densely connected QRNN encoder followed by a linear classification layer. The classification layer applies to the last hidden state of an encoded sequence.

Hyperparameters are the same as in the paper:

Parameter	Value
Hidden units	256
Num. layers	4
Embedding dim	300 (pretrained GloVe embedding)
Minibatch size	24
Dropout between layers	0.3
Pooling	ifo (not reported in paper)
Optimizer	RMSProp (learning rate = 0.001, $\alpha = 0.9$, $\epsilon = 10^{-8}$)
Regularisation	L^2 with weight=4×10^{-6}
Kernel size=Filter width	2 (except first layer)

Number of epochs and early stopping patience are not reported in the paper. I choose: Max. epochs= 50 and Patience = 5.

As in the paper, I train a model with kernel size 4 in the first layer and a model with kernel size 2 in the first layer.

Results Accuracy results are in Tab.1.

Model	Test set Accuracy
first layer kernel size = 2	88.36%
first layer kernel size = 4	87.12%

Table 1

I also give accuracy and loss curves for the model with kernel size 4 in the first layer in Fig. 1. Training time (50 epochs) for both models is approximately 19 hours on a single GPU.[4]

3.2 Language Modelling

This task is word-level language modelling.

Working on the Penn Treebank dataset, I find it very difficult to produce satisfying results with a QRNN language model. I observe 2 kinds of behaviour: Either the model gets stuck during early

[2]Available under http://ai.stanford.edu/~amaas/data/sentiment/ (last accessed 4th June 2020)

[3]Availabe under http://nlp.stanford.edu/data/glove.840B.300d.zip (last accessed 2nd July 2020)

[4]At this point of time, the implementation did not yet make use of the custom CUDA kernel. Perhaps it is faster now.

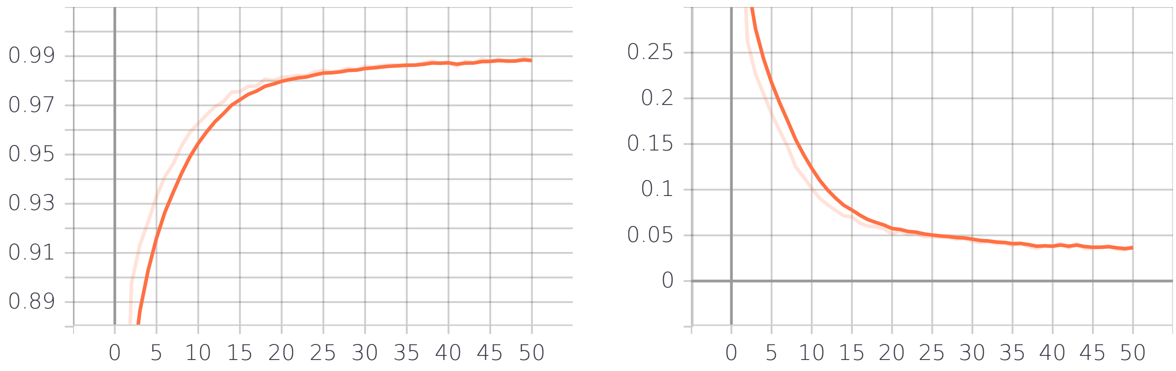

Figure 1: Movie review classification: Loss (right) and Accuracy (left) on the training set for all epochs (kernel size $= 4$). We see that the accuracy curve is approximating 1 and the loss curve is approximating 0. This shows that training happens.

training phase (this happens mostly when using SGD for optimization) or the model overfits the training dataset, resulting in very high perplexity on validation and test sets.

Also I would like to note that the problem remains when using the Salesforce QRNN implementation (together with the training procedure in AWD-LSTM-LM) despite differing claims by the authors.

Dataset Just as in the paper, I use the version of the Penn Treebank Dataset designed for language modelling from (Mikolov et al., 2010).[5] I do not apply any preprocessing except for adding special start and end tokens to every sentence. Also, I use whitespace tokenising, because the dataset is already prepared for language modelling (including tokenisation).

Model & Hyperparameters Since the model does not train either with the hyperparameters reported in the paper, nor with the hyperparameters suggested in the AWD-LSTM-LM repository, I tried out several combinations. The best result (listed below) was achieved by the following hyperparameters listed in Table 2.

Results This QRNN model achieves 108.86 perplexity on the test set. This is considerably worse than the ≈ 80 perplexity reported in the paper, and worse than the result of a LSTM language model using AWD-LSTM-LM.

The QRNN trained for 12 epochs, taking approximately 1 hour.

Parameter	Value
Hidden units	1240
Num. layers	4
Embedding dim	400
Minibatch size	20
Dropout between layers	0.4
Pooling	ifo
Zoneout	0.3
Grad norm	10
Dense convolutions	True
Optimizer	Adam (standard PyTorch parameters)
Kernel size=Filter width	2

Table 2: Hyperparameter for the QRNN language model

The setting would allow the model to train longer, but the validation perplexity starts to increase after the 3rd epoch, so early stopping (patience = 10) ends the training.

I give train set loss and perplexity curves in Fig. 2. Also, I give the validation perplexity in Fig. 3. Unfortunately, because of the strong increase in perplexity, only the values for the first epochs are visible. Validation perplexity for the last epoch is 469.83.

3.3 Character-level MT

Dataset Like in the paper, I use the IWSLT German-English dataset.[6] I prepared the data to be used with JoeyNMT and provide it together with

[5] Available under `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz` (last accessed 2nd July 2020)

[6] Available under `https://wit3.fbk.eu/download.php?release=2016-01&type=texts&slang=de&tlang=en` (Last accessed 2nd July 2020)

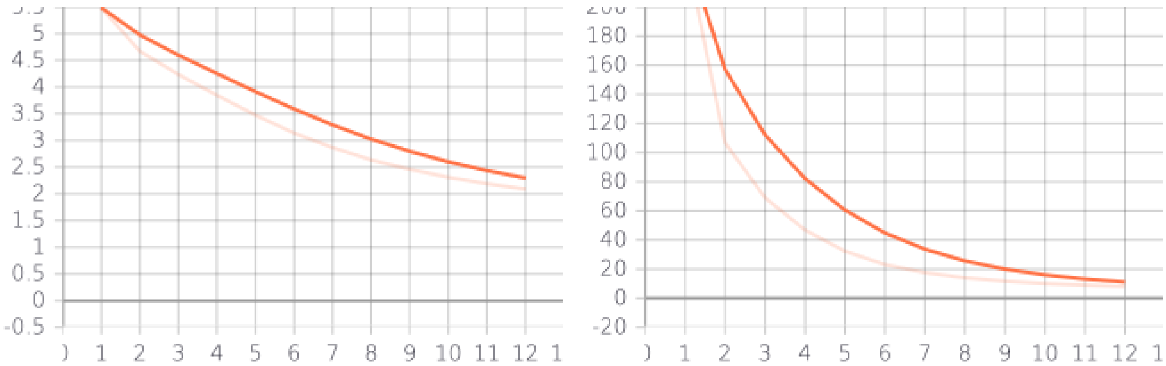

Figure 2: QRNN language model training on the Penn Treebank dataset for 12 epochs. Curves show train set loss (left) and train set perplexity (right)

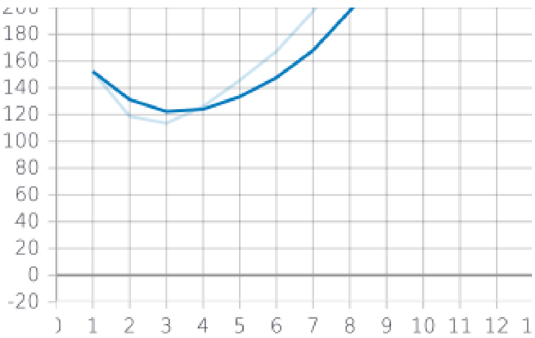

Figure 3: QRNN language model training on the Penn Treebank dataset for 12 epochs. The curve shows perplexity on the validation set.

Parameter	Value
Hidden units	320
Num. layers	4
Embedding dim	16
Minibatch size	16 (training) 10 (validation)
Dropout between layers	0.2
Dropout to attention	0.1
Pooling	ifo
Optimizer	Adam lr = 0.001, $\beta_1 = 0.9, \beta_2 = 0.999$
Regularisation	None
Kernel width	2 (first layer= 6)
Max seq. length	300
Beam size	8
Length penalty α	0.6

the code. The translation direction is from German to English. I don't lowercase characters. Maximum number of vocabulary items is 187, but the limit is not exhausted for both languages.

For training, I use the marked training data. For validation, I use the TED.tst2013 part and for testing the TED.tst2014 part. This is the same as in the paper.

Model & Hyperparameters For this task I used my implementation of the QRNN decoder. Encoder is the same custom QRNN implementation as for all other tasks. Hyperparameters closely match the hyperparameters stated in the paper:

Results The model was trained for 10 epochs ($\approx$ 12500 minibatches) without early stopping. This takes approximately 44 hours (1 day + 20 hours) on a single GPU. Test BLEU after training is 12.34.

My result is much worse than the result of 19.41 BLEU. However, my result is not representative, because due to the long training time, I did not perform hyperparameter tuning, so trained only once. Furthermore, my implementation is about 4 times slower than the speed reported in the paper. This could be due to the inefficient decoder, whose training cold be made faster by removing the step-by-step decoding procedure, so it can make full use of parallel computation of convolutions.

I provide perplexity and BLEU scores (beam size= 1) calculated on the validation data during training in Fig. 4. Furthermore, I show the attention visualisation of a validation sample in Fig. 5.

4 Conclusion

Some parts of reimplementing the QRNN (especially the features that are not available in the original implementation) have proven non-trivial (e.g. the custom CUDA-Kernel).[7] In particular, integrating the QRNN-Decoder variant into

[7] Admittedly, I have never worked with CUDA before.

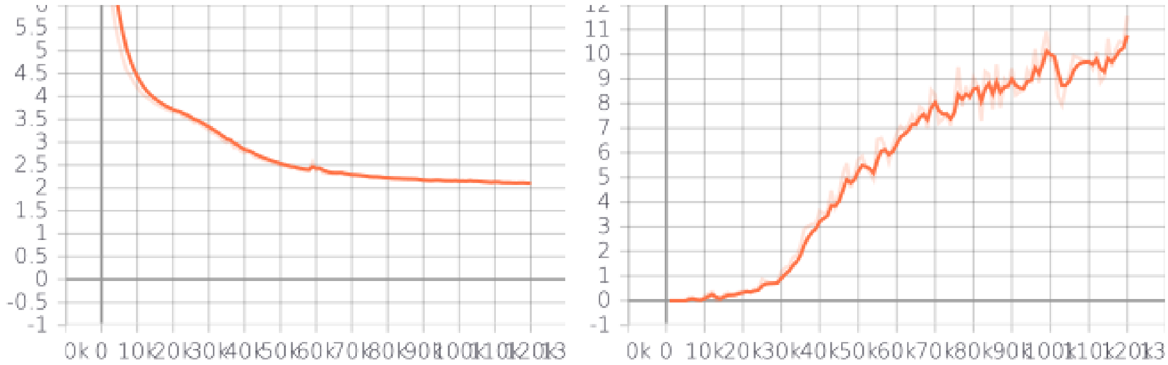

Figure 4: Validation perplexity (left) and validation BLEU (right) during training. For validating, greedy decoding (beam size 1) is used. Timesteps are minibatches.

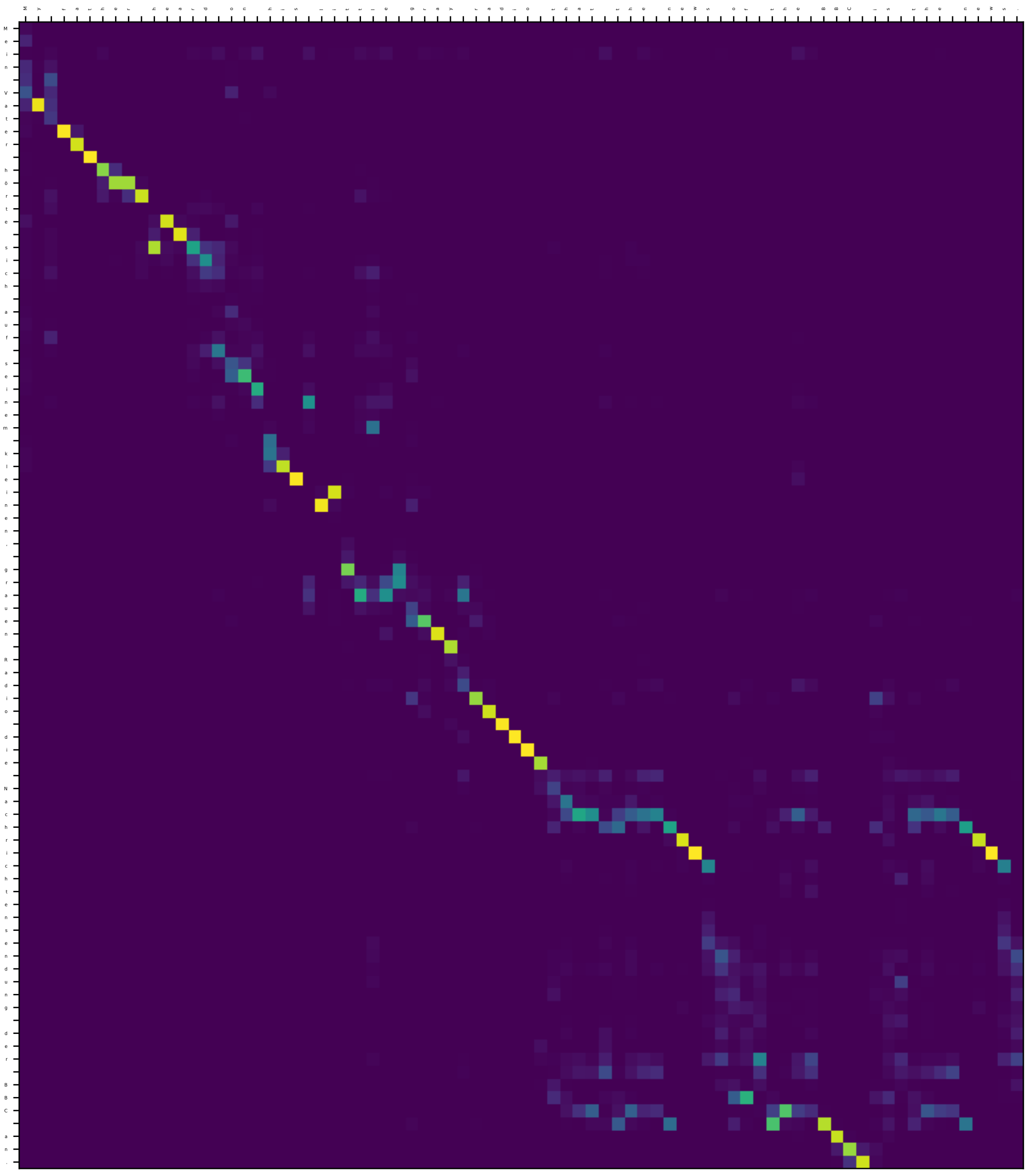

Figure 5: Attention visualisation of a sentence (training stage: minibatch nr. 12000, 10th epoch)

JoeyNMT was difficult, because using convolutions requires caching also the hidden states from previous timesteps, which is not natively supported by JoeyNMT. This shows that using non-standard architectures comes at the cost of higher implementation effort. This also concerns the claimed speedup in comparison to LSTMs. To truly reproduce the speedup, probably much more optimisation of the implementation would be necessary.

From a more theoretical perspective, it turns out that QRNNs seem to be more difficult to train than conventional LSTMs. In fact, I could not really reproduce results of any task described in the paper. As I have shown, the QRNN can be used to model the training set quite well, so the problem should not be an insufficient implementation, but rather problems in generalising to unseen data.

Concluding, I am not convinced of this kind of recurrent architecture, because the claimed speedup seems to be hard to achieve and the model's performance seems to be less secure than that of other types of RNNs.

References

James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. 2016. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*.

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.

Julia Kreutzer, Joost Bastings, and Stefan Riezler. 2019. Joey NMT: A minimalist NMT toolkit for novices. *To Appear in EMNLP-IJCNLP 2019: System Demonstrations*.

Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pages 142–150. Association for Computational Linguistics.

Tomas Mikolov, Martin Karafiat, Lukas Burget, Cernocky Jan, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (INTERSPEECH 2010)*, Makuhari, Chiba, Japan. `http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf`.

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 H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA. `http://is.muni.cz/publication/884893/en`.