Alma Mater Studiorum · University of Bologna

Question answering on the SQuAD dataset

NLP course final project

Leonardo Calbi (leonardo.calbi@studio.unibo.it)
Lorenzo Cellini (lorenzo.cellini3@studio.unibo.it)
Alessio Falai (alessio.falai@studio.unibo.it)

February 3, 2021

Contents

1 Summary

The tasks of Machine Comprehension (MC) and Question Answering (QA) have gained significant popularity over the past few years within the natural language processing and computer vision communities. Systems trained end-to-end now achieve great results on a variety of tasks in the text and image domains.

In this work, we address a question answering problem on the Stanford Question Answering Dataset (SQuAD) [21]: given a large collection of Wikipedia articles with associated questions, the goal is to identify the span of characters that contains the answer to the question.

This project focuses on the SQuAD v1.1 dataset, that contains more than 100 000 question-answer pairs on more than 500 articles. Here each question has at least one associated answer, whereas in the SQuAD v2.0 dataset there are also questions with no answer at all.

In this work we implement and compare five different models:

1. A simple LSTM encoder with a naïve attention mechanism, that will act as our Baseline
2. The BiDAF (Bi-Directional Attention Flow) network [20]
3. A model that wraps a pre-trained BERT (Bidirectional Encoder Representations from Transformers) language model [4]
4. A model that wraps DistilBERT [19], a distilled version of BERT
5. A model that wraps ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately) [3], a BERT model trained with a more sample-efficient pre-training task

Each and every model shares a custom output module, built specifically for question answering tasks, that outputs character-level spans representing answers in contexts, for each question in a mini-batch.

Recurrent-based models (i.e. Baseline and BiDAF) are trained from scratch, while Transformer-based models (i.e. BERT, DistilBERT and ELECTRA) are pre-trained on tasks different than question answering and fine-tuned on the SQuAD dataset. Our fine-tuning approach consisted in using a single small learning rate for both the backbone (i.e. the pre-trained model) and our output module, even though it could be beneficial to try the following procedure:

1. Start with a frozen backbone and train only the output module for a few epochs with a relatively high learning rate
2. Unfreeze the backbone and use progressive learning rates (smaller values near the start of the backbone and higher values towards the output layer)

Our experimental evaluations show that the best recurrent-based module is BiDAF, since it achieves 68% F1 and 55% EM on the validation split of the whole dataset, while for Transformer-based models ELECTRA shines as the best performing one, with around 84% F1 and 71% EM. Such results are on par with those reported in the corresponding models' papers.

2 Background

As already described above, the question answering task is based on the idea of identifying one possible answer to the given question as a subset of the given context.

Since the input data is of textual form and the latest models for the task are all based on neural architectures, there is the need to encode such text into a numeric representation. As of today, there are two main approaches to embed words in numerical format: sparse embeddings, like TF-IDF [17] and PPMI [12], and the modern dense embeddings, such as Word2Vec [10] and GloVe [16]. In the latter case, embeddings for each word in the input vocabulary are usually computed with shallow encoders.

These embeddings are then used as inputs for models specialized in processing sequential inputs. Nowadays, such models are mostly based on the Transformer architecture [23], which has the attention mechanism at its core. Instead, before this revolution, NLP competition's leaderboards were mostly populated by models based on recurrent and convolutional modules.

The most influential recurrent networks are LSTMs [8] and GRUs [5]. They are both based on processing sequential inputs and they keep an evolving series of hidden states, such that the output $h_t^{(0)}$ is a function of $h_{t-1}^{(0)}$ and the input x_t at position t. Recurrent layers can also be stacked to increase the capacity of the network: in that case, $h_t^{(0)}$ would act as an input to $h_t^{(1)}$, i.e. the first hidden state of the next depth-wise layer, and the same goes for the successive layers, as shown in figure 2.1.

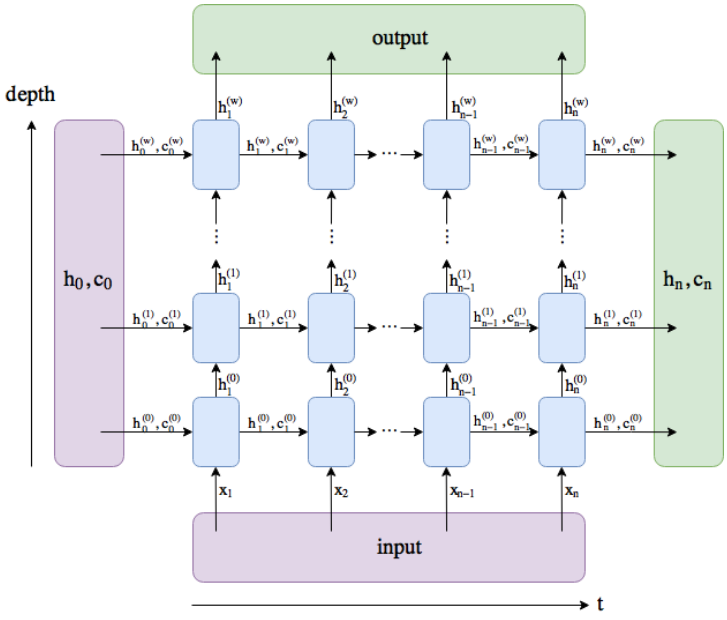

Figure 2.1: Recurrent module (image courtesy of [11])

In natural language, meaning does not usually simply flow from left to right, so that in RNNs other techniques may be necessary to reach a higher level of understanding. One example of such improvement is bi-directionality: the input is processed in two ways, from left to right (the "forward" pass) and from right to left (the "backward" pass); then, outputs of the two computations are merged together (usually by concatenating them over their last dimension). The intuition behind bi-directional RNNs is that they tend to understand the overall context better than their uni-directional counterparts, since they are able to aggregate information from the "past" and from the "future".

About Transformers, at a high-level, they comprise a stack of encoder-decoder modules, where each encoder features a multi-head attention block (to focus on different subsets of the input sequence when encoding one specific token) and a point-wise fully-connected layer, while each decoder features the same architecture as the encoder, but with an additional multi-head attention block in the middle (that helps the decoder focus on relevant parts of the input sequence).

The multi-head attention modules in the Transformer architecture are the ones responsible for gathering information from other tokens in the input sequence, thus resembling the task of hidden states in RNNs (i.e. incorporate the representation of previous inputs with the currently processed one).

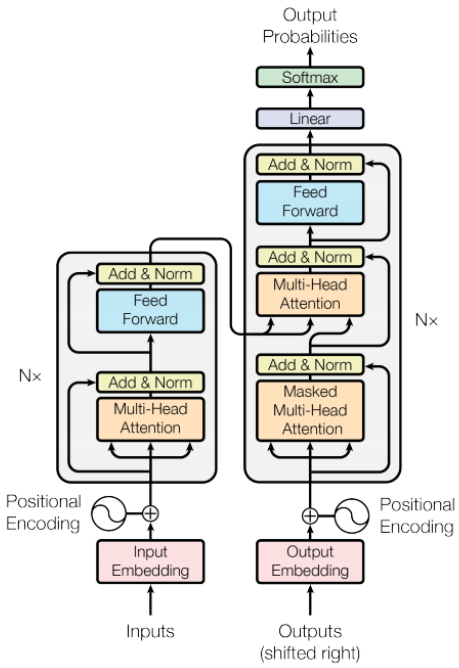

Figure 2.2: Transformers (image taken from [23])

While Transformers may always seem the way to go, since they throw away all the computational burden of training recurrent modules (which are sequential in nature and, thus, scarcely parallelizable), they also suffer from limitations due to the quadratic complexity of attention and the inherent maximum number of input tokens (which is implicitly linked to architecture choices themselves).

3 System description

In this work, we implemented five different models, grouped together in two categories:

- Recurrent-based models: comprising a simple LSTM encoder with a naïve attention mechanism (referred to as Baseline), along with the BiDAF network
- Transformer-based models: comprising wraps of language models, like BERT, DistilBERT and ELECTRA

In order to perform an effective models comparison, models from each group share the same input and output layer specifications.

The following is a detailed description of the models. Beware that each in each model section we report everything but what happens in the output module, which is described once in 3.3.

3.1 Recurrent-based models

3.1.1 Baseline

The following is a simple description of the flow followed by questions and contexts, when given in input to the Baseline model:

1. Embedding: it performs a standard word-level embedding on both questions and contexts (see 4.3)
2. Projection: embedded inputs are down-sampled into a fixed hidden size dimension, to reduce computation
3. Token encoding: a single uni-directional recurrent encoder, which processes questions and contexts as two separate inputs
4. Naïve attention: all the hidden states of a single question are averaged together (over the embedding dimension) so as to obtain a single vector which should encode the semantic information of the question at the sentence level
5. Query-aware context encoding: the aggregated question vector is element-wise multiplied to each context token latent representation

3.1.2 BiDAF

The BiDAF model [1, 20] is composed by the following sub-modules, where each one produces a more in-depth representation w.r.t. the previous stage, of the (question, context) tuple:

1. Embedding: simple word-level embeddings, even though the original BiDAF paper also leverages character-level embeddings (which are known to help with OOV words): given that this addition seems to increase the metrics only by a couple of percentage points while requiring a lot more computation and structure to work optimally, we decided to get rid of it

2. Projection: as done in the Baseline model
3. Highway: the last step in the embedding process is performed by a Highway network [22], which is usually used to balance the contributions of word and character embeddings, but in our case, lacking the character embedding part, it serves as a training stabilizer
4. Contextual embeddings: a single layer bidirectional LSTM is used to refine the word-level embeddings for both questions and contexts separately, by taking into account the surrounding words for each token
5. Attention: this step is the heart of the BiDAF model. First of all, it computes a similarity matrix between questions and contexts as $\langle w, [q, c, q \cdot c] \rangle$, where w is the weight matrix of a fully-connected layer, q and c represent the contextual embeddings of questions and contexts, respectively, $[\cdot]$ represents concatenation, $\langle \cdot \rangle$ is the dot product and $\cdot$ is element-wise multiplication. This similarity matrix represents how much each question and context words are grammatically and semantically alike and it's used as one of the inputs given to both BiDAF's peculiar attention modules, i.e. Context-To-Query (C2Q) and Query-To-Context (Q2C):
   (a) C2Q: represents which question words are most relevant to each context words, which outputs a vector $\tilde{c}$
   (b) Q2C: signifies which context words have the closest similarity to one of the question words and are hence critical for answering the query, which outputs a vector $\tilde{q}$

   The last step in the attention module outputs a query-aware context vector, given by $g = \langle w, [c, \tilde{c}, c \cdot \tilde{c}, c \cdot \tilde{q}] \rangle$, where w is th weight matrix of a fully-connected layer
6. Modeling: the modeling module is structured as a two-layer bidirectional LSTM, which takes as input the combined vector g

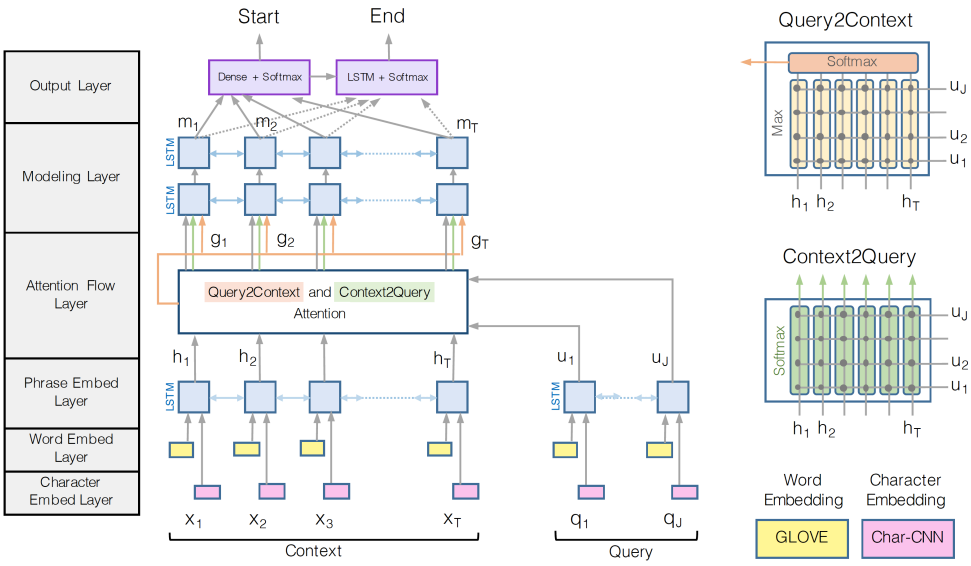

Figure 3.1: BiDAF model (image courtesy of [20])

3.2 Transformer-based models

3.2.1 BERT model

All the Transformer models are based on BERT [4], which is a language model composed by a multi-layer bidirectional encoder-decoder structure [23]. Our experiments are all performed using the base, uncased implementation of BERT, which has 12 Transformer layers and 768 as its hidden size.

The peculiarity of it's Transformer architecture allows the model to learn the context of a word based on all of its surroundings, at the same time, without having to perform a sequential scan of the input, as done in standard RNNs. This is reflected in its pre-training, which is composed of two tasks:

- Masked Language Modeling (MLM): at a high level, it is performed by masking 15% of the words in each sequence and replacing them with a $[MASK]$ token, so that the model is required to predict the original value of the masked words, using the available ones
- Next Sentence Prediction (NSP): achieved giving as input to the model a pair of sentences (formatted as in 4.2), which are in 50% of the cases two subsequent sentences or a random picked pair otherwise, so that the model is asked to identify if the second sentence follows the first one in the pair

The first task, MLM, is used to gain a detailed understanding of the processed language, while the second task, NSP, is used to let the model have a broader comprehension of semantic connections.

BERT is trained using both tasks together, with the goal of minimizing the combined loss.

3.2.2 DistilBERT model

BERT performances, as for the majority of large-scale language models, are inevitably intertwined with their number of parameters. DistilBERT [18, 19] tries to reduce this number using a technique called distillation, which consists in training a smaller "student" network to mimic the full output of the big "teacher" model [7]. Specifically, DistilBERT has about half the parameters of BERT and retains 95% of its performance on the GLUE benchmark [24].

3.2.3 ELECTRA model

ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately) [3] has the same architecture as BERT, but uses a new pre-training approach which aims to outperform the MLM strategy with RTD (Replaced Token Detection). Its structure is composed of three steps, shown in figure 3.2:

1. Given an input sequence, randomly replace tokens with a $[MASK]$ token
2. The generator (that performs a small MLM) predicts the original token for the masked one
3. The new sequence (with the replaced token) is given to the discriminator (ELECTRA), whose objective is to identify if each token is part of the original sequence or if it has been replaced by the generator

At the end of training the generator is not useful anymore, so that only the discriminator needs to be saved. This new pre-training task allows the model to compute the loss over each and every input token (not only on the masked ones, as done in BERT), which is what sets apart ELECTRA in terms of convergence speed and performance on downstream problems.

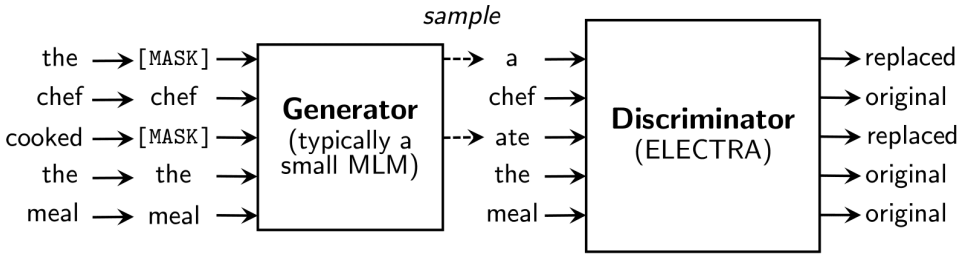

Figure 3.2: ELECTRA RTD pre-training task (image courtesy of [3])

3.3 Output module

The output module is composed by two fully-connected layers, used to identify the start and end token indexes of the answer in the given context. The start index classifier directly receives the output of the previous modules (e.g. in the BiDAF model that would correspond to the output of the modeling layer), while the same output vector is first passed into a single layer recurrent encoder before entering into the end index classifier. This additional RNN is used to implicitly enforce the constraint that the end index should follow the start one.

We then compute a softmax over both logits, so as to obtain a start and end index probability for each token in the context. To find the correct section of context that forms the answer, we compute the joint probability distribution (by multiplying each pair of start/end probabilities, for each token) and we take the start/end indexes corresponding to the maximum combined probability (respecting the assumption that the start index must precede the end one).

Finally, thanks to HuggingFace's tokenizers [9], we are able to simply convert start/end indexes over tokens to their corresponding values over original characters' positions.

The output layer is also responsible for the computation of the loss, which is a simple mean of cross-entropy losses of the start and end token probabilities.

4 Experimental setup and results

4.1 Data handling

Given the JSON training set, we decided to parse it into a suitable tabular data structure, which turned out to be useful even for subsequent pre-processing tasks. In particular, since each context has multiple question/answer pairs and each question can have multiple answers, we allocated one row for each context/question/answer triple, thus replicating the same context and question multiple times, in order to consider it as a separate example.

Regarding the splitting strategy for the dataset, we went for a simple holdout, thus setting aside 20% of the whole dataset for validation purposes. The way the validation set is built guarantees that the entire set of rows referencing to the same context is kept into the same split, thus avoiding unjustifiable boosting of results.

As an additional test, we evaluated all of our models on the official SQuAD v1.1 dev set. The main difference between this dataset and the training one is that the same question can have multiple correct answers.

Moreover, models are re-trained on the whole dataset (by merging training and validation splits), so as to simulate the cross-validation strategy. About re-training, results showed that this process only gives marginal improvements at the expense of a much more difficult way of choosing the best weights snapshots (since there are no validation metrics to observe, thus leaving only training loss available). Because of this, results reported in 4.6 do not show such training runs, but only their training/validation counterparts.

4.2 Tokenization

Tokenization has the following meaning: given a string of text, its task is to split it into substrings (called tokens), depending on the chosen algorithm. Different models call for different tokenization pipelines. In our case, we have two macro-categories:

1. Recurrent-based models tokenizer: splits words by whitespaces and punctuations, removes accents and applies a lowercasing function to all the tokens. Moreover, questions are padded (not truncated), while contexts are truncated and padded. Taking inspiration from [13], the maximum number of tokens for truncating contexts was fixed to 300.
2. Transformer-based models tokenizer: splits words using the WordPiece algorithm (introduced in [26]), normalizes unicode and foreign characters, removes accents and applies a lowercasing function to all the tokens, while also merging together questions and contexts as $[CLS]q_1q_2 \dots q_n[SEP]c_1c_2 \dots c_m[SEP]$ (it leverages the special tokens $[CLS]$ and $[SEP]$). Moreover, the combined question/context sentence is truncated to a maximum number of tokens (512, as in [4]) and padded to the right.

Tokenization happens on the fly at the batch level, thus enabling us to perform dynamic padding (based on the longest sequence in a batch) and avoiding the pre-tokenization overhead.

The tokenizer is also used as some kind of pre-processor, to remove from the training dataset those rows whose contexts would not contain the relative answer, after truncation.

4.3 Embeddings

In recurrent-based models, tokens are numerically encoded using the standard GloVe embeddings. In particular, we tested different types of GloVe models, with different embedding dimensions:

- Wikipedia 2014 and Gigaword 5 (6B tokens, 400K vocab, uncased, 300d vectors)
- Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 200d vectors)

We additionally embedded two more tokens (not present in the standard GloVe vocabulary):

1. $[PAD]$: the corresponding vector, which is used to pad sequences to a fixed lenght, contains all zeros
2. $[UNK]$: the corresponding vector, which is used to handle OOV (Out Of Vocabulary) words, is the mean of all the GloVe vectors, as reported by Pennington (one of GloVe's authors) [15]

   > *I've found that just taking an average of all or a subset of the word vectors produces a good unknown vector*

In Transformer-based models, we instead rely on the wrapped language models of the HuggingFace library [25], so that there is no need to manually handle token embeddings.

4.4 Metrics

Models are evaluated at the end of each epoch and metrics are computed for both the training and validation set. In our case, the validation set was mainly used to track models' performance on unseen data, in order to avoid overfitting problems.

Metrics used for evaluating the models are F1 and EM (Exact Match) and they were taken from the official SQuAD evaluation script. The F1 score ($f1 = \frac{2 \cdot p \cdot r}{p+r}$), defined as the harmonic mean between precision ($p = \frac{TP}{TP+FP}$) and recall ($r = \frac{TP}{TP+FN}$), is computed per question and then averaged across all questions. The EM score is instead computed as the number of questions that are answered in exact same words as the ground truth, divided by the total number of questions.

Since the same question can have multiple ground truths (specifically, in the dev set), there is the need to select just one for each question. To fulfill this requirement, only the nearest ground truth for each prediction is considered: given a prediction $p = [p_s, p_e]$, where p_s is the start index of the predicted answer and p_e is the corresponding end index, and a set of associated ground truths $G = \{[g_s^{(0)}, g_e^{(0)}], \ldots, [g_s^{(n)}, g_e^{(n)}]\}$, the selected ground truth is $\min_{g \in G} \|p - g\|_2$.

Moreover, we noticed that the training set does not have multiple answers for the same question, so this "nearest answer" approach comes into play only at inference time: this is actually the de-

sired behavior, since at training time the described technique could lead to an unstable learning process, given by the non-stationarity of the target distribution.

4.5 Environment

The main third-party libraries on which the project is based on are PyTorch [14], an open source machine learning framework with automatic differentiation and eager execution, and HuggingFace [25], a library to build, train and deploy state of the art NLP models.

All of the training and validation processes were executed on Google Colaboratory [6], a platform that gives the possibility to exploit some computational resources for free. In particular, Colab allows you to select a GPU runtime, that boosts the training time of neural models and most of the times we were assigned an NVIDIA Tesla T4 GPU, with 16GB of RAM.

Training and evaluation metrics, along with model checkpoints and results, are directly logged into a W&B project [2], which is openly accessible here.

4.6 Results

Table 4.1 shows the best results obtained in the training, validation and testing phases of each model, with the hyperparameters listed in table 4.2. In particular, by "training"/"validation" we mean metrics computed on the training/validation splits described in 4.1, while "testing" represents metrics obtained on the official SQuAD v1.1 dev set.

	Training		**Validation**		**Test**	
	F1 (%)	**EM** (%)	**F1** (%)	**EM** (%)	**F1** (%)	**EM** (%)
Baseline	36.21	21.27	37.19	26.47	33.69	21.70
BiDAF	63.42	43.27	68.40	55.31	71.15	60.09
BERT	73.68	56.14	80.28	67.80	83.17	74.17
DistilBERT	73.72	55.95	79.26	66.97	82.44	73.64
ELECTRA	76.42	58.89	84.45	71.83	88.27	80.60

Table 4.1: Best results

The hyperparameters listed in table 4.2 were mainly taken from the corresponding models' papers, in order to have a well-estabilished benchmark, even though some of them were adjusted based on available resources (e.g. the batch size).

	Epochs	**Batch size**	**Optimizer**	**Learning rate**
Baseline	30	128	Adam	$1e-3$
BiDAF	12	60	Adadelta	0.5
BERT	3	8	Adam	$5e-5$
DistilBERT	3	16	Adam	$5e-5$
ELECTRA	3	8	Adam	$5e-5$

Table 4.2: Hyperparameters

5 Analysis of results

6 Discussion

Bibliography

[1] Meraldo Antonio. *An Illustrated Guide to Bi-Directional Attention Flow (BiDAF)*. URL: https://towardsdatascience.com/the-definitive-guide-to-bi-directional-attention-flow-d0e96e9e666b.

[2] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from the site wandb.com. 2020. URL: https://www.wandb.com/.

[3] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. 2020. arXiv: 2003.10555 [cs.CL].

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. cite arxiv:1810.04805Comment: 13 pages. 2018. URL: http://arxiv.org/abs/1810.04805.

[5] Yuan Gao and Dorota Glowacka. "Deep Gate Recurrent Neural Network". In: *Proceedings of The 8th Asian Conference on Machine Learning*. Ed. by Robert J. Durrant and Kee-Eung Kim. Vol. 63. Proceedings of Machine Learning Research. The University of Waikato, Hamilton, New Zealand: PMLR, 2016, pp. 350–365. URL: http://proceedings.mlr.press/v63/gao30.html.

[6] Google. *Colaboratory*. URL: https://colab.research.google.com/.

[7] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: 1503.02531 [stat.ML].

[8] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[9] HuggingFace. *HuggingFace tokenizers*. URL: https://huggingface.co/docs/tokenizers/python/latest/.

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed Representations of Words and Phrases and their Compositionality". In: *NIPS*. Curran Associates, Inc., 2013, pp. 3111–3119. URL: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.

[11] Nikolai Morin. *What's the difference between hidden and output in PyTorch LSTM?* URL: https://stackoverflow.com/questions/48302810/whats-the-difference-between-hidden-and-output-in-pytorch-lstm.

[12] Yoshiki Niwa and Yoshihiko Nitta. "Co-Occurrence Vectors from Corpora vs. Distance Vectors from Dictionaries". In: *Proceedings of the 15th Conference on Computational Linguistics - Volume 1*. COLING '94. USA: Association for Computational Linguistics, 1994, pp. 304309. DOI: 10.3115/991886.991938. URL: https://doi.org/10.3115/991886.991938.

[13] Do-Hyoung Park and Vihan Lakshman. *Question Answering on the SQuAD Dataset*. Tech. rep. Stanford University.

[14] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[15] Jeffrey Pennington. *Unknown word tag*. URL: https://groups.google.com/g/globalvectors/c/9w8ZADXJclA/m/hRdn4prm-XUJ?pli=1.

[16] Jeffrey Pennington, Richard Socher, and Christopher Manning. "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://www.aclweb.org/anthology/D14-1162.

[17] "TF–IDF". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 986–987. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_832. URL: https://doi.org/10.1007/978-0-387-30164-8_832.

[18] Victor Sanh. *Smaller, faster, cheaper, lighter: Introducing DistilBERT, a distilled version of BERT*. URL: https://medium.com/huggingface/distilbert-8cf3380435b5.

[19] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *CoRR* abs/1910.01108 (2019). arXiv: 1910.01108. URL: http://arxiv.org/abs/1910.01108.

[20] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. "Bidirectional Attention Flow for Machine Comprehension". In: *CoRR* abs/1611.01603 (2016). arXiv: 1611.01603. URL: http://arxiv.org/abs/1611.01603.

[21] SQuAD. *SQuAD explorer*. URL: https://rajpurkar.github.io/SQuAD-explorer/.

[22] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. "Highway Networks". In: *CoRR* abs/1505.00387 (2015). arXiv: 1505.00387. URL: http://arxiv.org/abs/1505.00387.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[24] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". In: *CoRR* abs/1804.07461 (2018). arXiv: 1804.07461. URL: http://arxiv.org/abs/1804.07461.

[25] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *CoRR* abs/1910.03771 (2019). arXiv: 1910.03771. URL: http://arxiv.org/abs/1910.03771.

[26] Mike Schuster Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.