Sparse Word Representation for RNN Language Models on Cellphones

Chong Ruan[1,2]* and Yanshuang Liu[2]

[1] Schoold of Electronical Engineering and Computer Science, Peking University, Beijing 100871, China,
pkurc@pku.edu.cn

[2] Kika Tech, Longshaoheng Mansion, Hepingli South Road, Dongcheng District, Beijing 100013, China
yanshuang.liu@kikatech.com

Abstract. Language models are a key component of input methods, because they provide good suggestions for the next candidate input word given previous context. Recurrent neural network (RNN) language models are the state-of-the-art language models, but they are notorious for their large size and computation cost. A main source of parameters and computation of RNN language models is embedding matrices. In this paper, we propose a sparse representation-based method to compress embedding matrices and reduce both the size and computation of the models. We conduct experiments on the PTB dataset and also test its performance on cellphones to illustrate its effectiveness.

Keywords: language model, recurrent neural network, word embedding, sparse representation, input method

1 Introduction

Language modelling is a fundamental task in natural language processing, and can also be combined with other tasks such as spelling correction, machine translation and speech recognition. RNN language models [1,2] are capable of utilizing arbitrarily long history in theory, making them an ideal choice for language modelling. Despite RNN's powerful modelling capacity, its large size limits its application: the size of such models will easily grow to tens of megabytes or even larger, which is cumbersome for mobile or embedded devices.

Researchers have proposed many techniques to compress large neural networks, including weight pruning [3,4] and weight sharing [5,6]. Nevertheless, weight pruning leads to irregular connection patterns in the final pruned model, making it unfriendly to hardware; and weight sharing techniques often involve modifying standard neural network structures and sometimes impose extra constraints on training algorithms, making them difficult to incorporate with standard RNN layers. In [7] it has been observed that a large portion of parameters

* Corresponding author.

in RNN language models comes from the embedding matrix and proposed to compress the embedding matrix using sparse representation. Thus this method does not affect the network architecture and can be easily combined with popular RNN cells such as LSTM [8,9].

In this paper, we follow the same sparse representation approach as in [7] but go even further: we aim to reduce both the size and the computation cost of RNN language models. **Our main contributions are three-folds**:

- We propose a binary search procedure to ensure each vector is represented by a fixed number of basis vector, thus better exploits parallel processing capabilities of the hardware;
- We derive a re-formulation of logits computation which can be combined with sparse representation perfectly and reduces computation cost;
- We test our model's performance and availability on cellphones instead of just performing theoretical analysis.

The rest of this paper is organized as follows: In section 2, we review the background of our work. Our proposed method is illustrated in section 3 and experiment results are presented in section 4. Finally, section 5 concludes our work and discusses further directions.

2 Related Works

2.1 Skip-gram Word Vector

Word embeddings, also known as word vectors, are dense and low dimensional representations of words. One of the most popular tool to train word embeddings is *word2vec*[10]. We briefly summarize skip-gram model with negative sampling as follows:

$$L = \log \sum_{w \in C} \sum_{c \in Context(w)} g(w,c) = \sum_{w \in C} \sum_{c \in Context(w)} \log g(w,c) \tag{1}$$

$$g(w,c) = \log \sigma(E_w^T O_c) + \sum_{c_N \in NEG(w)} [\log \sigma(-E_w^T O_{c_N})] \tag{2}$$

where (w,c) is a word-context pair, $NEG(w)$ is the set of negative samples for word w, $\sigma(t) = 1/(1+\exp(-t))$ is the sigmoid function, $E, O \in \mathbb{R}^{n \times |V|}$ are input and output embedding matrices, and $E_x, O_y \in \mathbb{R}^n$ are input embedding for word x and output embedding for word y respectively. In many literature, only matrix E is regarded as word embeddings. However, as argued by [11], we use both matrix E and O in our experiment, which leads to a more effective way of initializing RNN language models.

2.2 RNN Language Model

RNN language models read a word embedding E_{w_t} as their input at each time step t, do some internal computation $f(\cdot)$, and predict the distribution of the next word $\boldsymbol{o}_t$:

$$\boldsymbol{h}_t = f(E_{w_t}, \boldsymbol{h}_{t-1}) \tag{3}$$

$$\boldsymbol{o}_t = softmax(P\boldsymbol{h}_t + \boldsymbol{c}) \tag{4}$$

, where $E \in \mathbb{R}^{n\times|V|}$ and $P \in \mathbb{R}^{|V|\times n}$ are input and output embedding matrix respectively, $\boldsymbol{c} \in \mathbb{R}^n$ is a bias term. In our experiment, we will use LSTM as the default RNN cell $f(\cdot)$.

2.3 Sparse representation

The sparse representation idea is to exploit the redundancy in embedding matrices: embedding matrix can be viewed as $|V|$ vectors in $\mathbb{R}^n$. Considering $|V| \gg n$, one can choose a group of over-complete bases in $\mathbb{R}^n$ and approximate each word vector using a sparse linear combination of basis vectors, which leads to a compression method. Since the number of parameters in embedding matrices is much larger than that in hidden layers ($O(n|V|)$ vs. $O(n^2)$), compressing embedding layers compresses the entire model.

The formulation in [7] is as follows: Denote basis matrix by $U \in \mathbb{R}^{|B|\times n}$ (each column of U is a basis vector in $\mathbb{R}^n$, and all $|B| > n$ columns form a group of over-complete bases), a word vector by $\boldsymbol{w}$, they determine basis weights $\boldsymbol{x} = (x_1, x_2, \cdots, x_{|B|})$ as the solution of the following optimization problem:

$$\min_{\boldsymbol{x}} \|U\boldsymbol{x} - \boldsymbol{w}\|_2^2 + \alpha\|\boldsymbol{x}\|_1 + \beta|\mathbf{1}^T\boldsymbol{x} - 1| + \gamma\mathbf{1}^T \max\{\mathbf{0}, -\boldsymbol{x}\} \tag{5}$$

, where the first term is the approximation error, the second term controls the sparseness of weights $\boldsymbol{x}$, the third term requires the sum of all weights to be close to 1, and the last term favors non-negative weights. While these regularization terms have their intuition, they introduce 3 additional hyper-parameters and make it more difficult to optimize.

They simply choose the word vectors of the most frequent words as the over-complete basis vectors and solve the equation above for all word vectors of infrequent words to obtain the sparse codebook. Because many components in $\boldsymbol{x}$ are zeros, one just need to store the indices and values of non-zero weights.

3 Methodology

This section consists of two subsections. In the first subsection, we describe the algorithm for learning sparse codings, the key part of which is a binary search procedure to ensure each sparse coding is of fixed length; in the second subsection, we illustrate how to utilize this sparse representation to reduce computation during prediction.

3.1 Proposed Sparse Representation

Our sparse representation technique is similar to the one in [7], but we simplify it to a basic LASSO problem and make it more tractable. We also assume words are sorted by their frequency in descending order, so that for a embedding matrix $E \in \mathbb{R}^{n \times |V|}$, its first $|B|$ columns $U = E_{1:|B|}$ are word vectors of the most frequent $|B|$ words, where $|B| > n$ is a hyper-parameter specified manually.

The proposed algorithm tries to represent a new word vector $\boldsymbol{w}$ by a linear combination of *exactly s* basis vectors. It has four inputs: an over-complete basis matrix $U \in \mathbb{R}^{n \times |B|}$, a new word vector $\boldsymbol{w}$ to be approximated, an integer s which indicates the desired sparseness, and a float tolerance *tol* used in terminating condition. And it returns two values: *indices*, an integer array of length s, denoting the ids of chosen basis vectors; and *weights*, a float array of length s, containing coefficients of the linear combinations.

The pseudo code of our algorithm is demonstrated in Algorithm 1.1. LASSO is used to control sparseness, but because we don't know the optimal regularization strength $\alpha*$ which can give us an exactly s-hot solution $\boldsymbol{x}*$, we set up a large range of the optimal $\alpha*$, and reduce this range by iterated trials. This binary search procedure converges very quickly.

When the range is small enough, we probably have obtained a good enough α_t, so we break it in line 13 and gather the indices and values of non-zero entries in current $\boldsymbol{x}*$. Note that there is a possibility that the number of non-zero entries in $\boldsymbol{x}*$ is slightly fewer than pre-specified s (because we break the loop in the strong regularization branch), we need to add more basis vectors with zero weights to embedding $\boldsymbol{w}$'s sparse representation to force a fixed length approximation, which is the "Zero padding" part of the algorithm.

For each column vector $E_j \in \mathbb{R}^n$ in the whole embedding matrix E, we can pass E_j as parameter $\boldsymbol{w}$ in algorithm 1.1 and learn a sparse representation for it. Run over all $|V|$ columns in E, we can get $|V|$ indices and weights and stack all them up into two matrices of shape $s \times |V|$, as illustrated in figure 1.

Compression ratio: The compression ratio is $n \times |V| / (n \times |B| + 2s \times |V|)$. Suppose $n = 400, s = 10, |B| = 2000$ and $|V| = 20000$, which is indeed a practical setting we used in our mobile experiment, the ratio is 6.67. Noting that the elements in index matrix I are all non-negative integers less than $|B|$, we can use even fewer bits to store this matrix. Output embedding matrix can be compressed similarly with each matrix in figure 1 transposed.

It is also very easy to restore a word vector from the basis vectors and its sparse representation: one just need to fetch proper basis vectors and add them up with corresponding weights, as in Algorithm 1.2.

3.2 Fast Prediction

To predict the next word, an RNN language model need to read current word embedding. For standard RNN language model, the current word embedding can be fetched directly from the input embedding matrix. For our sparse representation, one need to use algorithm 1.2 to recover the embedding and feed it

Algorithm 1.1 Sparse Code Learning Algorithm

```
1: procedure LEARN-SPARSE-CODING($U, \boldsymbol{w}, s, tol$)
        ▷ Choose $s$ vectors from columns of $U$ to approximate word embedding $\boldsymbol{w}$
2:     $\alpha_{min} \leftarrow$ 1e-3
3:     $\alpha_{max} \leftarrow$ 1e3
        ▷ Binary search
4:     while true do
5:         $\alpha_t = (\alpha_{min} + \alpha_{max})/2$
6:         $\boldsymbol{x}* \leftarrow \min_{\boldsymbol{x}} \frac{1}{2n} \|U\boldsymbol{x} - \boldsymbol{w}\|_2^2 + \alpha_t |\boldsymbol{x}|_1$
7:         $k \leftarrow$ NUMBER-OF-NON-ZERO-ENTRIES($x*$)
8:         if $k > s$ then        ▷ Regularization is too weak
9:             $\alpha_{min} \leftarrow \alpha_t$
10:        else        ▷ Regularization is too strong
11:            $\alpha_{max} \leftarrow \alpha_t$
12:            if $\alpha_{max} - \alpha_{min} < tol$ then
13:                break
14:            end if
15:        end if
16:    end while
        ▷ Extract non-zero entries from $\boldsymbol{x}*$
17:    indices $\leftarrow$ INDICES-OF-NON-ZERO-ENTRIES($\boldsymbol{x}*$)
18:    weights $\leftarrow$ VALUES-OF-NON-ZERO-ENTRIES($\boldsymbol{x}*$)
        ▷ Zero padding
19:    if $k < s$ then
20:        Randomly choose $s - k$ column ids from basis in $U$
21:        Append these $s - k$ ids to indices
22:        Append $s - k$ zeros to weights
23:    end if
        ▷ Now both indices and weights have exactly $s$ elements.
24:    return indices, weights
25: end procedure
```

Algorithm 1.2 Word Embedding Restoring Algorithm

```
1: procedure RESTORE-WORD-EMBEDDING($U$, indices, weights)
        ▷ Restore a word vector from its of sparse representation form
2:     $\boldsymbol{v} \leftarrow 0$
3:     for $i = 1..\text{len(indices)}$ do
4:         $\boldsymbol{v} \leftarrow \boldsymbol{v} + \text{weights}[i] U_{\text{indices}[i]}$        ▷ $U_j$ denote the $j$-th column of $U$
5:     end for
6:     return $\boldsymbol{v}$
7: end procedure
```

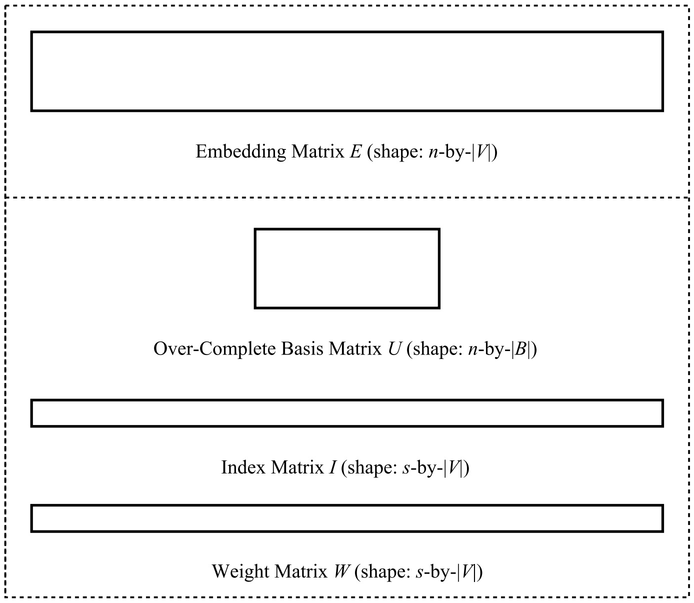
Embedding Matrix E (shape: n-by-|V|)
Over-Complete Basis Matrix U (shape: n-by-|B|)
Index Matrix I (shape: s-by-|V|)
Weight Matrix W (shape: s-by-|V|)

Fig. 1. Decompose embedding matrix into 3 smaller matrices

to RNN, which involves s embedding lookups, vector scaling and vector addition and thus increases model computation slightly. However, **we can use the shared bases to speedup the computation of the output side, a main source of computation cost in RNN language models, and reduce the total computation**.

Our goal is to calculate $P\boldsymbol{h}_t$ in equation 4 given a sparse decomposition of $P \in \mathbb{R}^{|V|\times n}$. Noting that the basis matrix is shared across all sparse representations, we can cache the product between basis matrix $\hat{U}$ and hidden state $\boldsymbol{h}_t$ based on the following key observation:

$$\langle P_i, \boldsymbol{h}_t \rangle = \left\langle \sum_{j=1}^{s} \text{weights}[j] \hat{U}_{\text{indices}[j]}, \boldsymbol{h}_t \right\rangle = \sum_{j=1}^{s} \text{weights}[j] \left\langle \hat{U}_{\text{indices}[j]}, \boldsymbol{h}_t \right\rangle \quad (6)$$

, where *weights* and *indices* are sparse representation for P_i.

Denote the sparse decomposition of $P \in \mathbb{R}^{|V|\times n}$ by $\hat{U}, \hat{I}, \hat{W}$ (they are of shape $|B|$-by-n, $|V|$-by-s, and $|V|$-by-s respectively, not to be confused with input embedding matrix E's decomposition U, I, W in figure 1), we have the following fast multiplication algorithm 1.3:

Reduction in flops: The original $P\boldsymbol{h}_t$ requires $n|V|$ float multiplications and $(n-1)|V|$ additions, while algorithm 1.3 requires only $n|B| + s|V|$ multiplications and $(n-1)|B| + (s-1)|V|$ additions. Again, if $n = 400, s =$

Algorithm 1.3 Fast Multiplication Algorithm

```
1: procedure FAST-MULTIPLICATION($\hat{U}, \hat{I}, \hat{W}, \boldsymbol{h}_t$)
                       ▷ Compute $P\boldsymbol{h}_t$ from a sparse decomposition of $P$, i.e.: $\hat{U}, \hat{I}, \hat{W}$
2:     $\boldsymbol{v} \leftarrow \hat{U}\boldsymbol{h}_t$                                ▷ $\boldsymbol{v}$ is of length $|B|$
3:     $\boldsymbol{r} \leftarrow \mathbf{0}_{|V|}$                          ▷ $\boldsymbol{r}$ is a zero-vector of length $|V|$
4:     for $i = 1..|V|$ do
5:         $r_i \leftarrow \sum_{j=1}^{s} \hat{W}_i[j] v_{\hat{I}_i[j]}$     ▷ $\hat{W}_i, \hat{I}_i$ denote the $i$-th row of $\hat{W}, \hat{I}$ respectively
6:     end for
7:     return $\boldsymbol{r}$
8: end procedure
```

$10, |B| = 2000$ and $|V| = 20000$, we reduce the computation cost by a factor of $(n|V| + (n-1)|V|)/(n|B| + s|V| + (n-1)|B| + (s-1)|V|) = 8.08$.

4 Experiments

In this section we show our experiment results on PTB dataset[3] and model performance on cellphones. To recap, our model is basically an RNN language model described in subsection 2.3, where the input embedding E_{w_t} in equation 3 is computed using algorithm 1.2 and $P\boldsymbol{h}_t$ in equation 4 is computed with algorithm 1.3. We use LSTM as the default RNN cells.

We pretrain input and output embeddings with skip-gram models using python package *gensim* [12] and draw 5 negative samples for each word in training data. The input and output embedding matrices in this paper corresponds to member variables *syn0* and *syn1neg* in class *gensim.models.KeyedVectors* respectively. Then we use algorithm 1.1 to decompose pretrained embedding matrices and initialize parameters of our sparse RNN language model. For parameters within hidden layers, we simply initialize them from uniform distribution as in [2]. The index matrices I and $\hat{I}$ are fixed thereafter, while basis matrices and weight matrices are kept finetuned during the training phase, which is another difference from paper [7]. Training is performed using *TensorFlow*[13].

4.1 PTB

For PTB dataset, we set up two experiments: small and large. All our hyperparameters and training protocols follow [2][4]. Both the small model and large model have a vocabulary size of 10,000 and 2 LSTM layers. However, the hidden size is different: 200 for small model and 1,500 for large model. The perplexity results are reported in table 1 (the lower, the better):

We see that our sparse model has a higher training perplexity, but the gap between train and test perplexity is smaller. Actually, the sparse constraints

[3] Available at `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz`
[4] See `https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py` for details.

Table 1. Perplexity on PTB dataset

Model		train	test
Small	standard [2]	37.99	115.91
	sparse ($s = 10, \|B\| = 2k$)	69.67	**110.88**
Large	standard [2]	37.87	**78.29**
	sparse ($s = 20, \|B\| = 4k$)	55.23	82.35

act as a regularizer and prevent overfitting. We see that small sparse model performs better that the standard model, but large sparse model is worse. This phenomenon can be explained by the ratio $|B|/n$: for the small model, the bases are more over-complete ($|B|/n = 2000/200 = 10$), and the sparse approximation is pretty precise; while for the large one, the ratio is only $4000/1500 = 2.67$, which leads to some approximation error and causes degeneracy in performance.

4.2 Performance on Cellphones

In this part, we compare 3 different models. The first model is a standard RNN language model, with embedding size 400 and 2 LSTM hidden layers of size 400. The second one uses algorithm 1.1 to compress the output embeddings and algorithm 1.3 to speedup prediction. The third model further compresses the input embedding matrix.

The models are trained on an internal corpus, which is consisting of 10M sentences and the domain is daily conversation. We normalize all punctuation to <pun> and numbers to <num>, and keep the most frequent 20,000 words in the final vocabulary. For sparse representation, we set basis size $|B| = 2000$ and sparseness $s = 10$.

On a Macbook Pro Retina 2015, we test the memory consumption and response time (time for inference 1 step) of these 3 models, and summarize the results in table 2. The response time is the average value of 200 inferences.

Table 2. Memory and response time on Macbook

Model	Model1: basic	Model2: sparse softmax	Model3: sparse
Memory consumption (MB)	72	47	24
Response time (ms)	16.5	9.5	7.5

From model 1 to model 3, we can see memory does reduce a lot due to our compression algorithm. And there is a large drop in response time from model 1 to model 2, which is the effect of our fast multiplication algorithm. Surprisingly, the response time of model 3 is even shorter than that of model 2, which can be attributed to better locality and modern cache system.

We also test our models on a Nexus 5, and we don't know any previous work that has tested their method on real cellphones. The memory consumption is roughly the same as that on the Macbook, so we omit it here. We run 100 predictions for each model, and list the response time as follows:

Table 3. Response time on Nexus 5

Model	Model1: basic	Model2: sparse softmax	Model3: sparse
Response time (ms)	30∼33	15∼28	19∼22

It's clear that the last two models are faster, and the third model is more stable than the second one.

We also compare our model's performance by combining it with LatinIME, an open source input method editor. The default language model of LatinIME is based on n-gram models, and it integrates unigram to trigram counts with a complicated algorithm and empirical values. When predicting next word, it utilizes previous context and current incomplete character sequence. Character sequence is fed to an internal Trie tree to find words with similar spellings, and the language model reranks the candidate words based on previous context. We replace the n-gram language model with our RNN language model, and compare the input efficiency of these two methods. **The input efficency is defined as the ratio of number of real characters to that of keystrokes**. For example, if a user wants to input the word "happy", and he managed to achieve this by typing only the first 3 letters "hap" (because input method recommend the word "happy" to him), the input efficiency is len("happy")/len("hap") = 5/3. The test result is in table 4.

Table 4. Input efficiency statistics

Method	LatinIME	LatinIME with RNNLM
Input efficiency	1.55	1.83

We see that LatinIME with RNNLM behaves significantly better than original LatinIME with n-gram language model. Actually, we do observe bad predictions of the original LatinIME like "in two days ago", because it thinks both "in two days" and "two days ago" are valid. When combined with RNNLM, these bad cases rarely occur.

5 Future Works

In this paper, we propose a method to decompose a word embedding matrix into an over-complete basis matrix, an index matrix, and a weight matrix. By fixing the length of each sparse coding and computing the logits of next word distribution though the linear combination of those of output basis vectors, we reduce both the size and computation cost of the model and make it more suitable to run on hardware.

In the future, we plan to explore other kinds of redundancy, e.g.: sharing the input and output over-complete basis matrices, tying input and output embedding matrices, etc. These techniques will make the model smaller and more efficient.

References

1. Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.
2. Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.
3. Abigail See, Minh-Thang Luong, and Christopher D. Manning. Compression of neural machine translation models via pruning. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016, Berlin, Germany, August 11-12, 2016*, pages 291–301, 2016.
4. Sharan Narang, Gregory Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.
5. Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.
6. Zhiyun Lu, Vikas Sindhwani, and Tara N. Sainath. Learning compact recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, pages 5960–5964, 2016.
7. Yunchuan Chen, Lili Mou, Yan Xu, Ge Li, and Zhi Jin. Compressing neural language models by sparse word representations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.
8. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
9. Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.
10. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
11. Ofir Press and Lior Wolf. Using the output embedding to improve language models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017, Valencia, Spain, April 3-7, 2017, Volume 2: Short Papers*, pages 157–163, 2017.
12. Radim Řehůřek and Petr Sojka. 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, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.
13. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.