MJEN

MANAS Journal of Engineering
ISSN 1694-7398 | e-ISSN 1694-7398

Volume 9, Issue 1, (2021) Pages 15-21
https://doi.org/10.51354/mjen.822630

A TensorFlow implementation of Local Binary Patterns Transform

Devrim Akgun

Sakarya University, Software Engineering Department, 54187, Sakarya, Turkey, dakgun@sakarya.edu.tr, ORCID: 0000-0002-0770-599X

ABSTRACT

Feature extraction layers like Local Binary Patterns (LBP) transform can be very useful for improving the accuracy of machine learning and deep learning models depending on the problem type. Direct implementations of such layers in Python may result in long running times, and training a computer vision model may be delayed significantly. For this purpose, TensorFlow framework enables developing accelerated custom operations based on the existing operations which already have support for accelerated hardware such as multicore CPU and GPU. In this study, LBP transform which is used for feature extraction in various applications, was implemented based on TensorFlow operations. The evaluations were done using both standard Python operations and TensorFlow library for performance comparisons. The experiments were realized using images in various dimensions and various batch sizes. Numerical results show that algorithm based on TensorFlow operations provides good acceleration rates over Python runs. The implementation of LBP can be used for the accelerated computing for various feature extraction purposes including machine learning as well as in deep learning applications.

ARTICLE INFO

Research article

Received: 6.11.2020
Accepted: 28.06.2021

Keywords:
tensorflow,
local binary patterns,
deep learning,
feature extraction

1. Introduction

Intense computations in Python may take considerably longer time when compared with the programming languages such as C/C++. This is because Python is an interpreter based language where the input script is interpreted line by line. On the other hand Python provides a high level abstraction which makes most of the scientific computations and especially machine learning and deep learning applications simple to implement. Although Python realization of these algorithms usually take long running times, C/C++ compiled Python functions accelerates the computations significantly. For this purpose TensorFlow provides an open source framework for machine learning applications [1], [2]. TensorFlow also enables programmers to run their codes on GPU (Graphics Processing Unit) and TPU (Tensor processing Unit) as well as on CPU. Hence utilization of GPU resources and hardware accelerators in addition to CPU provides significant accelerations [3]. TensorFlow library includes most of the functions and layers for developing and training deep learning models. In addition, it has the flexibility of custom layers which enable programmers to design their own layers [4]. Users may describe custom functions based on the existing operations as well as writing the functions from scratch.

In deep learning applications, determining the layers of deep neural networks have importance in developing a successful model. Deep learning layers have the ability to extract features from the dataset automatically, and different layers may provide the potential to extract better features. Feature extraction layers such as convolutions and pooling in deep learning enable to automatic extraction of desired features. Although TensorFlow and Keras cover most of the frequently used layers, additional layers may increase the accuracy depending on the problem's nature. Various authors use custom layers as a combination of existing layers or their own developed layers for specific purposes such as new activation function definitions for medical diagnostic [5], wavelet-based pooling [6], solution of inverse partial differential equation [7], radial basis functions for adaptive routing problems [8].

One of the feature extraction methods is LBP transform which is widely used in machine learning and deep learning applications in addition to image processing [9], [10]. In literature there are numerous utilizations of LBP in various computer vision applications such as handwritten text recognition [11], facial expression recognition for smart applications [12], ear recognition for identity verification [13], Retrieval of histopathological image retrieval [14],

gender recognition [15], edge detection for noisy images [16], breast tumor diagnosis [17], texture image retrieval [18], face similarity comparison [19], color texture recognition [20]. Most deep learning or machine learning models for computer vision applications like image LBP demand intense computational power. In this study, a general-purpose LBP operation is written using basic TensorFlow operations. Experimental evaluations were realized using image batches ranging from 1 to 1024 and images in various dimensions ranging from 28×28 to 448×448. In order to show the acceleration of the Tensorflow based algorithm, the results were also obtained in Python. Proposed design can be used as a layer of a deep learning model as well as general purpose image processing applications. The rest of the paper was organized as follows; a brief information about LBP was given in Section 2, and the proposed design with TensorFlow was explained in Section 3. Comparative evaluations with the TensorFlow model were done in Section 4. Conclusions about the evaluations were given in the final section.

2. Local binary patterns

Deep learning models are intended to automatically extract the features required to make correct estimations. In computer vision, deep learning models, 2D convolution layers, and pooling layers are the key operators for automatic feature extraction. There are also preprocessing layers such as normalization, noise reduction, and histogram equalizations for the elaboration of the training dataset to provide better accuracy. One of the efficient feature extraction approaches is LBP transform [21], which extracts the texture features efficiently in pattern recognition studies. It can be used in deep learning applications for preprocessing or a non-trainable layer for increasing the model accuracy depending on the problem type.

$$LBP_{K,R}(i,j) = \sum_{k=0}^{K-1} f(p_k, p_c) 2^k \qquad f(p_c, p_n) = \begin{cases} 1 & p_c < p_n \\ 0 & otherwise \end{cases} \tag{1}$$

The idea of LBP transform is shown by Eq. 1, where K is the number of neighbor pixels, R is the radius, p_k is one of the selected neighbor pixels, and p_c is the center pixel of the selected window. An example application of this equation for K=9 and R=3, which corresponds to a 3×3 window for the computation of each pixel, is given in Figure 1. The first pixel that is used for comparison is selected as the Least Significant Bit (LSB), and it forms the first digit of the binary number, which corresponds to 2^0. In this example, it is selected as p_0=187, and when compared with the pixel at the center p_c=191, $p_0 \geq p_c$ produces "0". Similar operations are repeated clockwise till all comparisons are made for the remaining pixels. If all comparisons are written as a binary digit, it is obtained as; (01011000)2=88, the corresponding LBP transform value for the selected pixel. When the described operations are repeated for all pixels, the LBP transform is obtained in the form of a 3×3 matrix. Note that the input dimensions can be maintained by using padding to the input matrix. An example application of LBP for a complete image is shown in Figure 2.

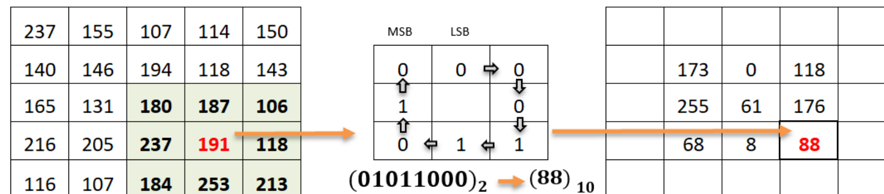

Figure 1. *An example image input image on the left and its LBP transform output on the right*

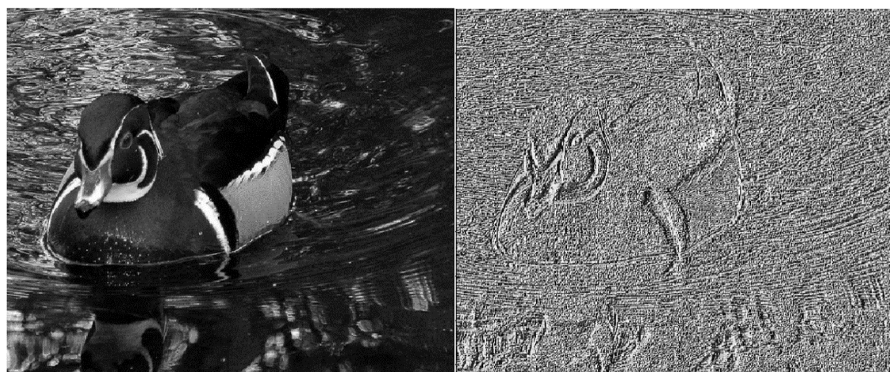

Figure 2. *An example image input image on the left and its LBP transform output on the right*

3. TensorFlow implementation

TensorFlow is an open-source numerical computation library for machine learning and deep learning applications. It has support for various high-level programming languages such as Python, C++, and Java. TensorFlow also has a low-level API to communicate with various hardware, as shown by the hierarchical block diagram given in Figure 3. There are defined implementations such as layers, losses, metrics, and various TensorFlow operations on the top of low-level API. TensorFlow provides various operations for the execution of algorithms on multicore CPU and GPU resources and makes it practical for general-purpose computations as well as training deep learning models. Data and variables in TensorFlow are defined by tensors which are N-dimensional arrays in Python. In TensorFlow 1, a session is started to compute library operations for given tensor data and variables. Recently it has been made more practical with the introduction of TensorFlow 2, removing the need for a session by running Eager execution by default.

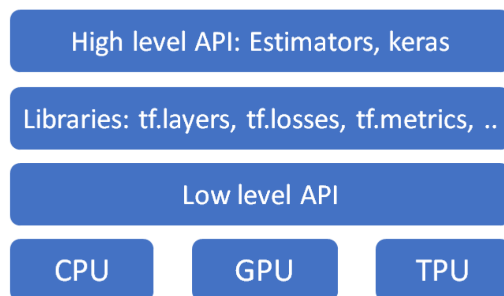

Figure 3. *Hierarchical structure of TensorFlow framework*

```
L=np.zeros((rows,cols,1),np.float32)
for i in range(1,rows-1):
    for j in range(1,cols-1):
        L[i,j]=\
        ( I[i-1,j]  >= I[i,j] )*1+\
        ( I[i-1,j+1]>= I[i,j] )*2+\
        ( I[i,j+1]  >= I[i,j] )*4+\
        ( I[i+1,j+1]>= I[i,j] )*8+\
        ( I[i+1,j]  >= I[i,j] )*16+\
        ( I[i+1,j-1]>= I[i,j] )*32+\
        ( I[i,j-1]  >= I[i,j] )*64+\
        ( I[i-1,j-1]>= I[i,j] )*128;
```

Code snippet 1. Python implementation of LBP transform

The TensorFlow framework provides various operations that can operate on tensors such as *add*(), *matmul*(), *mean*(), and *greate*r(), and combinations of these can be used to write new operations. In the LBP algorithm that is straightforward to implement, various comparison, multiplication, and adding operations are used as shown by Code snippet 1. According to the algorithm, all pixels are computed independently, and these can be defined with tensor operations. A matrix based implementation of this algorithm is given in Figure 4. Since TensorFlow operations are mainly defined for vector-matrix operations, most of the for-loops that can be defined in parallel are eliminated. In this implementation, the input variables; $P0$, $P1$, …, $P7$ define the neighbors in selected 3×3 mask in the form of matrices. The pixels in the selected masks are compared with the pixels at the center of masks one by one, as previously described in Eq. 1. After a comparison is made, false and true conditions are defined in the form of a matrix and then the comparison is done. This is repeated for every eight different pixels in a mask to form the LBP transform of the image.

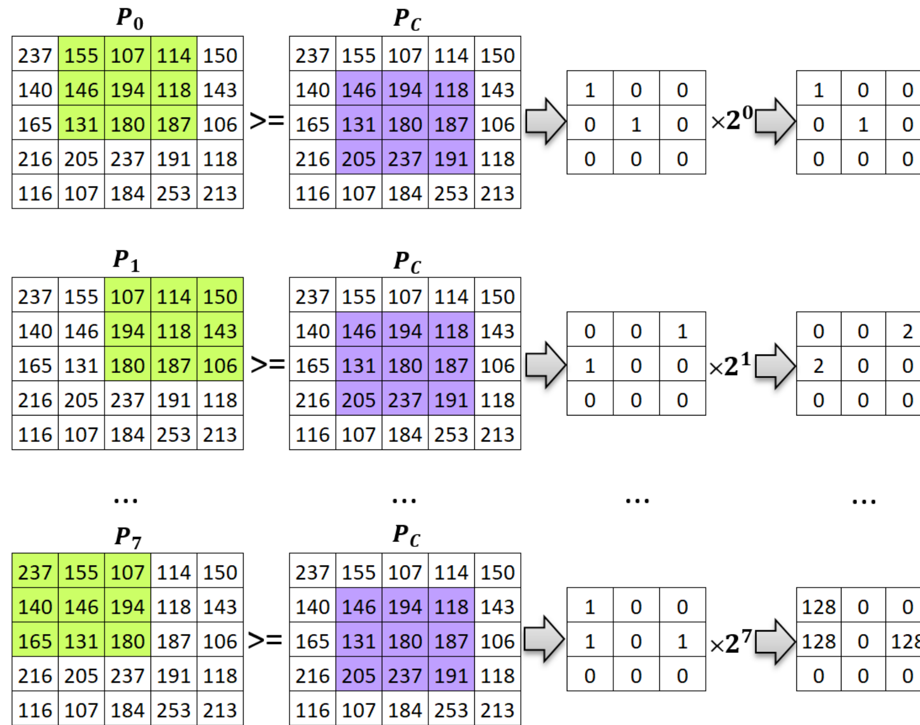

Figure 4. *TensorFlow based implementation approach*

```
g=tf.greater_equal(y01,y11)
z=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(1,dtype='uint8') )
g=tf.greater_equal(y02,y11)
tmp=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(2,dtype='uint8') )
z =tf.add(z,tmp)
g=tf.greater_equal(y12,y11)
tmp=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(4,dtype='uint8') )
z =tf.add(z,tmp)
g=tf.greater_equal(y22,y11)
tmp=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(8,dtype='uint8') )
z =tf.add(z,tmp)
g=tf.greater_equal(y21,y11)
tmp=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(16,dtype='uint8') )
z =tf.add(z,tmp)
g=tf.greater_equal(y20,y11)
tmp=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(32,dtype='uint8') )
z =tf.add(z,tmp)
g=tf.greater_equal(y10,y11)
tmp=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(64,dtype='uint8') )
z =tf.add(z,tmp)
g=tf.greater_equal(y00,y11)
tmp=tf.multiply(tf.cast(g,dtype='uint8'), tf.constant(128,dtype='uint8') )
z =tf.add(z,tmp)
```

Code snippet 2. TensorFlow implementation of LBP transform

A TensorFlow implementation of this algorithm is given by Code snippet 2. Since TensorFlow operations are mainly defined for vector-matrix operations, most of the for-loops defined in parallel are eliminated. The pixels in the selected masks are compared with the pixels at the center of masks one by one, as previously described in Eq. 1. After a comparison is made, false and true conditions are defined in the form of a matrix, and then the comparison is made. This is repeated for every eight different pixels in a mask to form the LBP transform of the image.

4. Experimental results

Experimental evaluations were realized using Python 3.7.9 and TensorFlow 2.3.1 on Ubuntu 18.04 operating system. Test hardware has an AMD FX2700 eight-core CPU and GTX1080 GPU which has 8GB of memory and 2560 CUDA cores. Time measurements for the test runs were realized with the time library of the python as given by the example code shown by Code snippet 3. All the measurements were repeated 30 times to and the average time is used to form experimental results. Image batches were selected from the ImageNet data set [22], and the sizes of the images were resized to test cases which range from 28×28 to 448×448 for all evaluations. However, it should be noted that the contents of the images usually affect the extracted features but have ignorable effects on the execution durations that are hard to measure.

```
# get current time
start_time = time.time()

#Compute LBP using TensorFlow for given batch of images
result = tf_lbp(batch_of_images).numpy()

# get current time and compute the elapsed time
elapsed_time  = time.time() - start_time
```

Code snippet 3. TensorFlow implementation of LBP transform

Table 1. *Python implementation running times (seconds) for LBP algorithm*

Batch size	Image size				
	28×28	**56×56**	**112×112**	**224×224**	**448×448**
1	0.0187	0.0774	0.3172	1.2609	4.9940
2	0.0365	0.1533	0.6269	2.5125	10.107
4	0.0713	0.3009	1.2458	5.0312	20.010
8	0.1425	0.5987	2.4868	10.000	40.354

Table 1 shows the results for the Python implementation for comparison with the TensorFlow results. Numerical results show that CPU running time varies somewhat in proportion to the image size, which is a usual case. As the image size is increased, running time is increased in the same way. For example, while a 224×224 image is processed in 1.26 seconds, a 448×448 image is processed in 4.99 seconds. Because the number of pixels is increased four times, the running time is also increased approximately four times. Similar behavior is observed when the number of images in the batch is increased, as shown by Figure 5, where speed-up evaluations are given. This is not the case for GPU running times when the numerical results given in Table 2 given for TensorFlow are investigated. This is due to the cost of initializing the GPU devices and the management overhead of the CUDA cores.

Table 2. *TensorFlow running times (seconds) for LBP algorithm*

Batch size	Image size				
	28×28	**56×56**	**112×112**	**224×224**	**448×448**
1	0.0017	0.0018	0.0019	0.0020	0.0023
2	0.0017	0.0018	0.0019	0.0021	0.0029
4	0.0017	0.0018	0.0020	0.0023	0.0039
8	0.0018	0.0018	0.0020	0.0027	0.0064
16	0.0018	0.0019	0.0022	0.0038	0.0116
32	0.0019	0.0020	0.0025	0.0063	0.0266
64	0.0019	0.0022	0.0036	0.0109	0.0787
128	0.0020	0.0027	0.0064	0.0211	0.1703
256	0.0020	0.0032	0.0082	0.0566	0.2341
512	0.0021	0.0033	0.0082	0.0563	0.2349
1024	0.0021	0.0033	0.0086	0.0577	0.2385

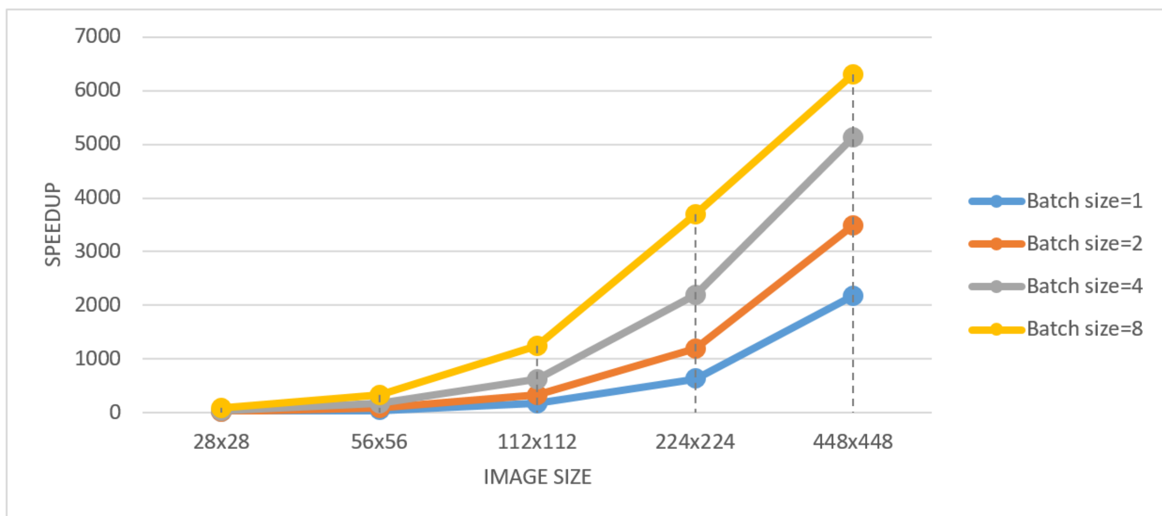

Figure 5. *TensorFlow speed-up over Python implementation*

5. Conclusions

In the presented study, a method for accelerating the LBP transform computations using TensorFlow operators has been proposed. The LBP algorithm has been defined in terms of matrix operations so that TensorFlow operators can be efficiently used. The acceleration provided by the Tensorflow method has been illustrated by comparing it with baseline Python implementation. The results show that TensorFlow running times for the LBP algorithm are far better than the direct Python runs. The significant difference between the running times of Python and TensorFlow algorithms is mainly due to the two factors. First, python codes are interpreted line by line, and therefore, results are computed slower than compiled codes. The other is GPU acceleration provided by TensorFlow library in addition to using compiled functions. Speed-up obtained by TensorFlow increases considerably as the image size is increased. This is because initializations and GPU device communications for small operations are usually costly. As the image size increases, the overhead of managing CPU and GPU device gets smaller, resulting in more efficiency. Designed LBP algorithm can be used to accelerate computer vision applications that involve LBP transform since TensorFlow allows general-purpose computations.

References

[1] Abadi M. *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” *arXiv Prepr. arXiv1603.04467*, (2016).

[2] Abadi M.*et al.*, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, (2016), 265–283.

[3] Chien S.W.D., Markidis S., Olshevsky V., Bulatov Y., Laure E., Vetter J., "TensorFlow Doing HPC," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (2019), 509–518.

[4] Agrawal A.*et al.*, "TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning," *arXiv Prepr. arXiv1903.01855*, (2019).

[5] Parisi L., Neagu D., Ma R., Campean F., "QReLU and m-QReLU: Two novel quantum activation functions to aid medical diagnostics," *arXiv Prepr. arXiv2010.08031*, (2020).

[6] Williams T., Li R., "Wavelet pooling for convolutional neural networks," in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, (2018).

[7] Pakravan S., Mistani P.A., Aragon-Calvo M.A., Gibou F., "Solving inverse-PDE problems with physics-aware neural networks," *arXiv Prepr. arXiv2001.03608*, (2020).

[8] Perepelkin D., Ivanchikova M., "Research of Neural Network Architectures for Solving Adaptive Routing Problems in Multiprovider Networks of Distributed Data Centers," in *2020 9th Mediterranean Conference on Embedded Computing, MECO 2020*, (2020), 1–5.

[9] Pietikäinen M., Hadid A., Zhao G., Ahonen T., *Computer Vision Using Local Binary Patterns*, vol. 40. Springer Science & Business Media, (2011).

[10] Pietikäinen M., "Local Binary Patterns," *Scholarpedia*, 5, 3, (2010), 9775.

[11] Al-Shatnawi A., Al-Saqqar F., Alhusban S., "A holistic model for recognition of handwritten arabic text based on the local binary pattern technique," *Int. J. Interact. Mob. Technol.*, 14, 16, (2020), 20–34.

[12] Nigam S., Singh R., Misra A. K., "Local Binary Patterns Based Facial Expression Recognition for Efficient Smart Applications," in *Security in Smart Cities: Models, Applications, and Challenges*, Springer, (2019), 297–322.

[13] Hassaballah M., Alshazly H.A., Ali A.A., "Ear recognition using local binary patterns: A comparative experimental study," *Expert Syst. Appl.*, vol. 118, (2019), 182–200.

[14] Erfankhah H., Yazdi M., Babaie M., and Tizhoosh H.R., "Heterogeneity-Aware Local Binary Patterns for Retrieval of Histopathology Images," *IEEE Access*, vol. 7, (2019), 18354–18367.

[15] El-Alfy E.S.M., Binsaadoon A.G., "Automated gait-based gender identification using fuzzy local binary patterns with tuned parameters," *J. Ambient Intell. Humaniz. Comput.*, 10, 7, (2019), 2495–2504.

[16] Shen T., Huang F., Jin L., "An improved edge detection algorithm for noisy images," *ACM Int. Conf. Proceeding Ser.*, 36, 3, (2019), 84–88.

[17] Touahri R., Azizi N., Hammami N.E., Aldwairi M., Benaida F., "Automated breast tumor diagnosis using local binary patterns (LBP) based on deep learning classification," in *2019 International Conference on Computer and Information Sciences, ICCIS 2019*, (2019), 1–5.

[18] Yang W., Krishnan S., "Combining Temporal Features by Local Binary Pattern for Acoustic Scene Classification," *IEEE/ACM Trans. Audio Speech Lang. Process.*, 25, 6, (2017), 1315–1321.

[19] Gundogdu B., Bianco M. J., "Collaborative similarity metric learning for face recognition in the wild," *IET Image Process.*, 14, 9, (2020), 1733–1739.

[20] Hosny K.M., Magdy T., Lashin N.A., "Improved color texture recognition using multi-channel orthogonal moments and local binary pattern," *Multimed. Tools Appl.*, (2021), 1–16.

[21] Ojala T., Pietikäinen M., Mäenpää T., "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns," *IEEE Trans. Pattern Anal. Mach. Intell.*, 24, 7, (2002), 971–987.

[22] Deng J., Dong W., Socher R., Li L.-J., Kai Li, Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," (2010), 248–255.