

A PyTorch Operations Based Approach for Computing Local Binary Patterns

Devrim Akgun




Department of Software Engineering, Faculty of Computer and Information Sciences, Sakarya University, 54050 Sakarya, Turkey (dakgun@sakarya.edu.tr) ORCID [0000-0002-0770-599X](https://orcid.org/0000-0002-0770-599X)

Abstract

Advances in machine learning frameworks like PyTorch provides users with various machine learning algorithms together with general purpose operations. PyTorch framework provides Numpy like functions and makes it practical to use computational resources for accelerating computations. Also users may define their custom layers or operations for feature extraction algorithms based on the tensor operations. In this paper, Local Binary Patterns (LBP) which is one of the important feature extraction approaches in computer vision were realized using tensor operations of PyTorch framework. The algorithm was written both using Python code with standard libraries and tensor operations of PyTorch in Python. According to experimental measurements which were realized for various batches of images, the algorithm based on tensor operations considerably reduced the computation time and provides significant accelerations over Python implementation with standard libraries.

Author Keywords. PyTorch, Local Binary Patterns, Feature Extraction, Machine Learning.

Type: Research Article

 Open Access  Peer Reviewed  CC BY

1. Introduction

Machine learning applications usually involves special preprocessing for feature extraction and these can have significant effect on the success of the designed model. Especially developing a deep learning model may require defining custom layers for better feature extraction and training the model usually involve large datasets. When Python script with standard libraries are used for writing programs computation times may increase considerably, depending on the size of the dataset and the type of the processed data. Because Python is an interpreter based language, evaluation of the program script line by line usually increases the computation time specifically for loop operations. Various libraries like PyTorch are provided to make computations faster for machine learning algorithms as well as practical. PyTorch frameworks includes compiled algorithms mainly designed for machine learning and deep learning applications such as natural language processing, sequence processing and computer vision. Using the tensors of PyTorch frame work for writing a custom operation eliminates, if possible, most of the loops and it enables efficient utilization of computational resources. Custom operations can be used to build custom layers in machine learning and deep learning applications ([Paszke et al. 2019](#)). Various researchers design their frameworks, custom layers, custom loss functions or custom operations based on the available PyTorch operations such as a kernel design for xnor and bitcount computations ([Xu](#)

and Pedersoli 2019), Bayesian optimization framework in PyTorch (Balandat et al. 2019), normalized convolutional neural network (Kim et al. 2020).

Some of the deep learning applications requires preprocessing of special layers for feature extraction that are not exist in the algorithms of PyTorch library. In such cases programmers may define their custom layers based on the present operations without writing a tensor operation from scratch. In computer vision applications, one of the important feature extraction approaches is the LBP transform (Pietikäinen et al. 2011; Pietikäinen 2010). Due to efficient and computationally lightweight structure, it has been used in various machine learning applications such as LBP network for face recognition (Xi et al. 2016), facial expression recognition (Rahul, Kohli, and Agarwal 2020), image forgery detection (Alahmadi et al. 2017), acoustic scene classification (Yang and Krishnan 2017), dermoscopic skin lesion image segmentation (Pereira et al. 2020), Hyperspectral image classification (Tu et al. 2019), on-road vehicle detection in urban traffic scene (Hassaballah, Kenk, and El-Henawy 2020).

Computer vision algorithms like LBP usually requires high CPU usage due to various matrix multiplications and additions. In the cases of machine learning or deep learning model developments, computational power need increases further as the dataset and the image dimensions to be processed are increased. In this work, a custom function for the accelerating the computation of LBP transform was implemented based on the available PyTorch operations. Various dimensions of images up to 448×448 were used in the performance evaluations. Also the effect of batch size were evaluated by changing the number of images in test batches up to 256. In order to obtain acceleration results, LBP algorithm was also implemented using standard Python functions. In the following section, the details about LBP transform and PyTorch implementation for the LBP were introduced. Experimental running time evaluations were presented in Section 3. Discussions and concluding remarks were given in the final section.

2. Materials and Methods

2.1. Local binary patterns

LBP is an efficient as a pattern descriptor that has been used in various computer vision applications (Ojala, Pietikäinen, and Mäenpää 2002) including machine learning and deep learning applications. LBP transform involves checking the neighbors of a pixel within a given radius with the center pixels. According to the results of each comparison, an LBP value is formed as a result. A general representation for LBP transform for a given pixel is given by Formula 1. In this equation, R is the radius of the area, K is the number of neighbors, p_c and p_k represents the center pixel and a selected neighbor pixel respectively. The function $f(.)$ represents a comparison function. In the application of this equation, sum operator selects a neighbor pixel and compare it with the pixel at the center. If the pixel at the center is smaller, corresponding value is set to 2^k otherwise it is set to zero. This is repeated for all neighbors to complete transform.

$$LBP_{K,R}(i,j) = \sum_{k=0}^{K-1} f(p_k, p_c) 2^k \quad (1)$$

$$f(p_c, p_n) = \begin{cases} 1 & p_c < p_n \\ 0 & otherwise \end{cases}$$

In the present study, 3x3 window which is the common case in practice was used. Hence, the number of neighbors was set to 9 and radius was set to 3. Figure 1 shows an example computation for a selected pixel. Once a neighbor is selected for comparison, the result is added to sum. According to formula, the first neighbor forms the Least Significant Bit (LSB) and the last neighbor forms the Most Significant Bit (MSB) when the result is considered in binary form. This operation is repeated for all pixels in order to obtain LBP transform. An example image and its LBP equivalent are shown by Figure 2.

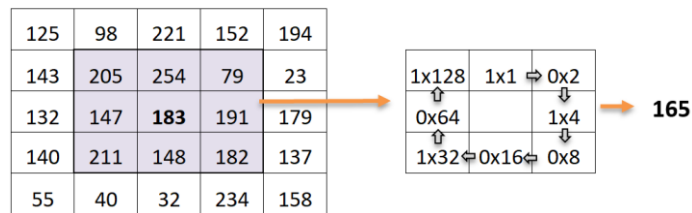


Figure 1: An example image input image on the left and its LBP transform output on the right

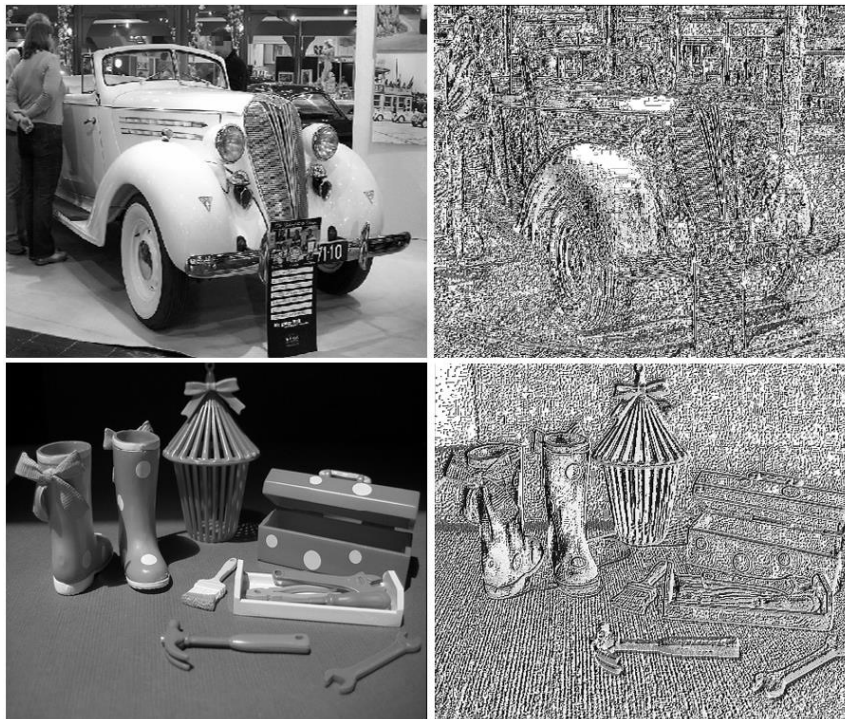


Figure 2: Example images on their visualized LBP transform outputs

2.2. PyTorch implementation

Python is one of the mostly preferred programming languages for machine learning and scientific computations. On the other hand, Python programs usually take long running times when the costly algorithms written using scripts instead of compiled functions. PyTorch is one of the popular frameworks that provides various accelerated algorithms for developing machine learning and deep learning applications. It is designed to work with tensor based algorithms that enables accelerated computation. As its name implies PyTorch is primarily designed for Python programming language, but it also provides language bindings for Java

and C++ as well as working on various operating systems as shown by Figure 3 where the hierarchical structure of the framework is given.

PyTorch Build	Stable (1.7.1)		Preview (Nightly)		
Your OS	Linux	Mac	Windows		
Package	Conda	Pip	LibTorch	Source	
Language	Python		C++ / Java		
CUDA	9.2	10.1	10.2	11.0	None

Figure 3: Hierarchical structure of PyTorch (Paszke et al. 2019)

Data types in PyTorch are defined by multi-dimensional matrices which are called as tensor. These enables the utilization of optimized algorithms that work, if exist, on multicore CPU and GPU devices. There are various tensor operation such as *add()* for adding tensors, *mul()* for multiplying each element of the tensor and *ge()* for greater and equal comparison. These can be used to design custom operations for specific purposes. LBP algorithm contains independent pixels operations and these can be computed using matrix definitions instead of using two for-loop to travel through the pixels of the image. The center and neighbor pixels can be selected in the form of matrices as described by Figure 4. Therefore total nine matrix which are P1,...,P9 and PC are used to compute Formula 1 at one time. The comparison operations are all realized in the matrix form as well as addition and multiplication operations. Element wise comparison of selected area for the center pixel matrix with the selected areas for neighbor matrices leads to matrices of comparison results in the form of ones and zeros as shown by Figure 5. This comparison in PyTorch can be realized with *ge()* operator.

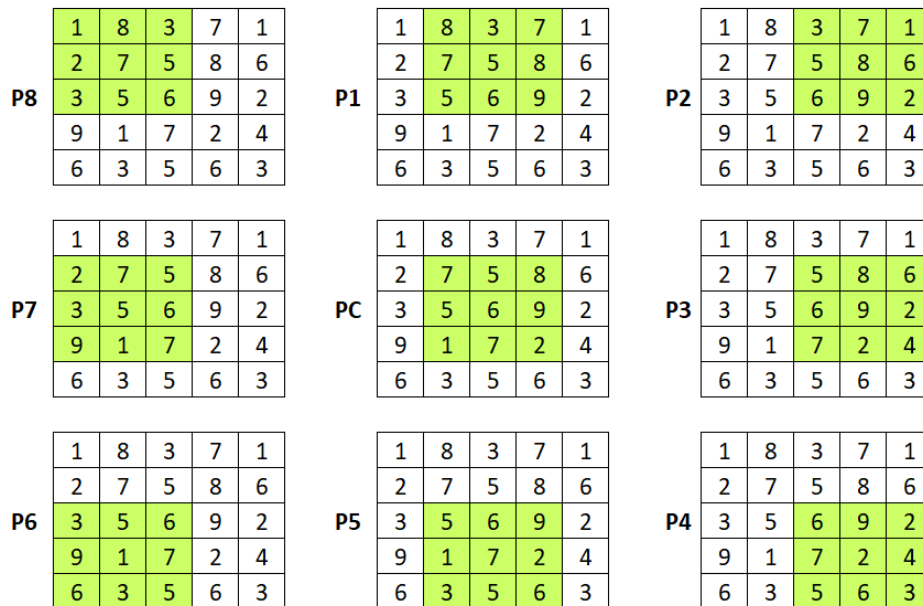


Figure 4: Neighbor selection approach on the example input

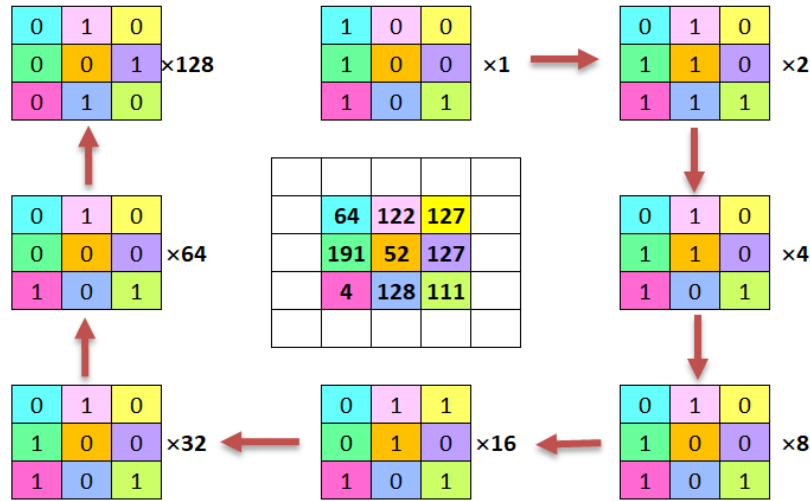


Figure 5: Comparison results with the PC matrix and computing LBP values

After the first comparison is done, the resulting matrix is multiplied with 2^0 as previously given by Formula 1 and then the result is written to a temporary variable. Similarly, after the second comparison and multiplication obtained results is accumulated in the temporary variable. Following computations for the other pixels are realized in the same way and the results accumulated in the temporary variable for the LBP transform. Element wise multiplications and addition of the resulting matrices can be done using *mul()* and *add()* tensors respectively. A verification test was done by comparing the results of each method for a given random input matrix as shown console outputs given by Figure 6. For both approach the input matrices are zero padded to maintain image size. According to outputs, tensor based approach produces the desired results.

```

Random test numbers:
[[237 107 105 210 174 32 180]
 [129 173 188 19 48 175 96]
 [193 174 207 70 180 157 197]
 [197 216 120 106 200 191 42]
 [247 109 32 203 217 161 164]
 [ 33 226 165 205 70 71 192]
 [ 72 134 141 238 192 109 178]]

Python computation result:
[[ 0 120 116  0 72 124  0]
 [ 29 188 18 255 189 42 113]
 [ 24 118 32 252 24 117  0]
 [ 20 32 201 222 48 98 241]
 [ 0 219 255 20  0 205 144]
 [ 31 128 78 18 255 191  0]
 [ 6  7 135  0 192 70  1]]

PyTorch computation result:
[[ 0 120 116  0 72 124  0]
 [ 29 188 18 255 189 42 113]
 [ 24 118 32 252 24 117  0]
 [ 20 32 201 222 48 98 241]
 [ 0 219 255 20  0 205 144]
 [ 31 128 78 18 255 191  0]
 [ 6  7 135  0 192 70  1]]
    
```

Figure 6: Example verification results using Python and PyTorch for a given random matrix. The input is zero padded to protect dimensions

3. Experiments

Performance measurements were done on a hardware that has an Intel(R) Core(TM) i7-4710MQ which is a four-core CPU with eight threads. Python 3.7.4 and PyTorch 1.3.0 on Windows 10 operating system. Running times of the algorithm were measured with the *time()* function of time library in Python. Image dimensions were selected as 28×28, 56×56, 112×112, 224×224 and 448×448 which are close to the dimensions used in deep learning applications.

Also the size of test batches were selected as 2^n where n is varied between 0 and 7. Images for performance measurements were examined on images used from ImageNet [19] dataset and these images were scaled to test dimensions in each case. Although there may be small differences for the computation of Formula 1 due to if-else blocks, the number of pixel operations are independent from the pixel contents. Hence, contents of the images have trivial impact on the processing times for images in the same dimensions and the same size of batches are close to each other.

	28×28	56×56	112×112	224×224	448×448
1	0.0159	0.0646	0.2526	1.0367	4.1352
2	0.0319	0.1269	0.5021	2.0516	8.3094
4	0.0643	0.2559	1.0145	4.1181	16.540
8	0.1281	0.5022	2.0409	8.1713	32.913
16	0.2598	1.0312	4.0453	16.371	65.848
32	0.5040	1.9953	8.1727	33.205	131.30
64	1.0207	4.0368	16.189	69.582	265.92
128	2.0119	8.0721	32.520	131.26	530.73
256	4.0422	16.050	64.610	261.32	1053.8

Table 1: Python implementation running times (seconds)

	28×28	56×56	112×112	224×224	448×448
1	0.0003	0.0005	0.0009	0.0010	0.0218
2	0.0004	0.0006	0.0014	0.0020	0.0460
4	0.0005	0.0009	0.0012	0.0201	0.0878
8	0.0006	0.0014	0.0022	0.0442	0.1774
16	0.0009	0.0013	0.0215	0.0847	0.3539
32	0.0015	0.0030	0.0434	0.1790	0.6909
64	0.0016	0.0206	0.0835	0.3405	1.3521
128	0.0023	0.0433	0.1710	0.6759	2.5744
256	0.0212	0.0854	0.3585	1.3467	4.9751

Table 2: PyTorch running times (seconds)

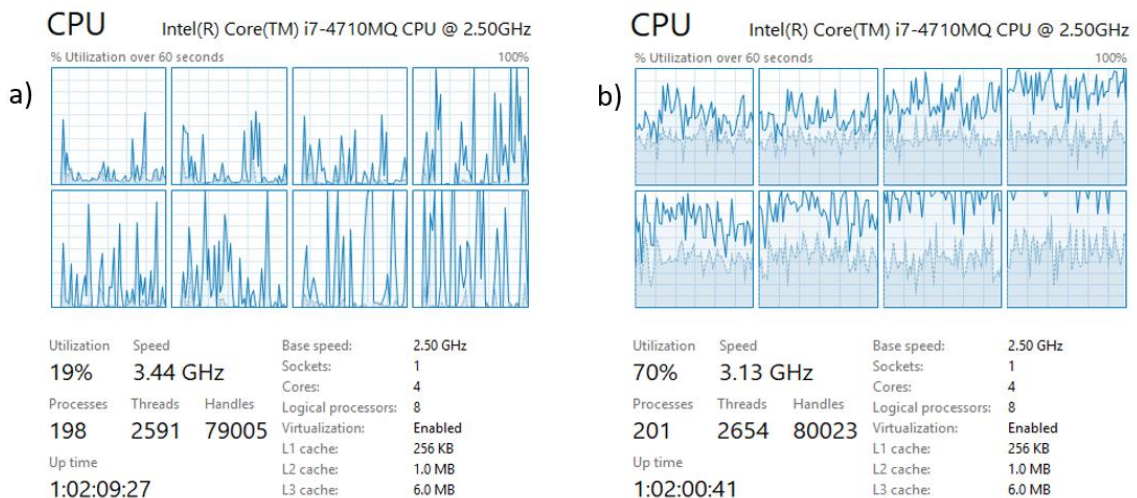


Figure 7: CPU utilization graphs for a) Python and b) PyTorch

Table 1 shows example execution durations for Python implementation with standard libraries. For small sizes of images and small sizes of batches, the running times measured are considerably small. As the number of images or the image dimensions are increased, the execution times are also increased significantly. In the case of PyTorch based implementation, computation times are considerably reduced as shown by Table 2. When the CPU utilization graphs are examined, PyTorch significantly increase the CPU utilization when compared with the standard implementation as shown by Figure 7. The speedup results provided by PyTorch over Python implementation with standard libraries are given by Figure 8. The acceleration is mainly the result of compiled code and multicore CPU utilization. Differences among the speedup results for different batch sizes and image sizes are mainly depends on the management of threads, behavior of multicore CPU and the number of pixels operations.

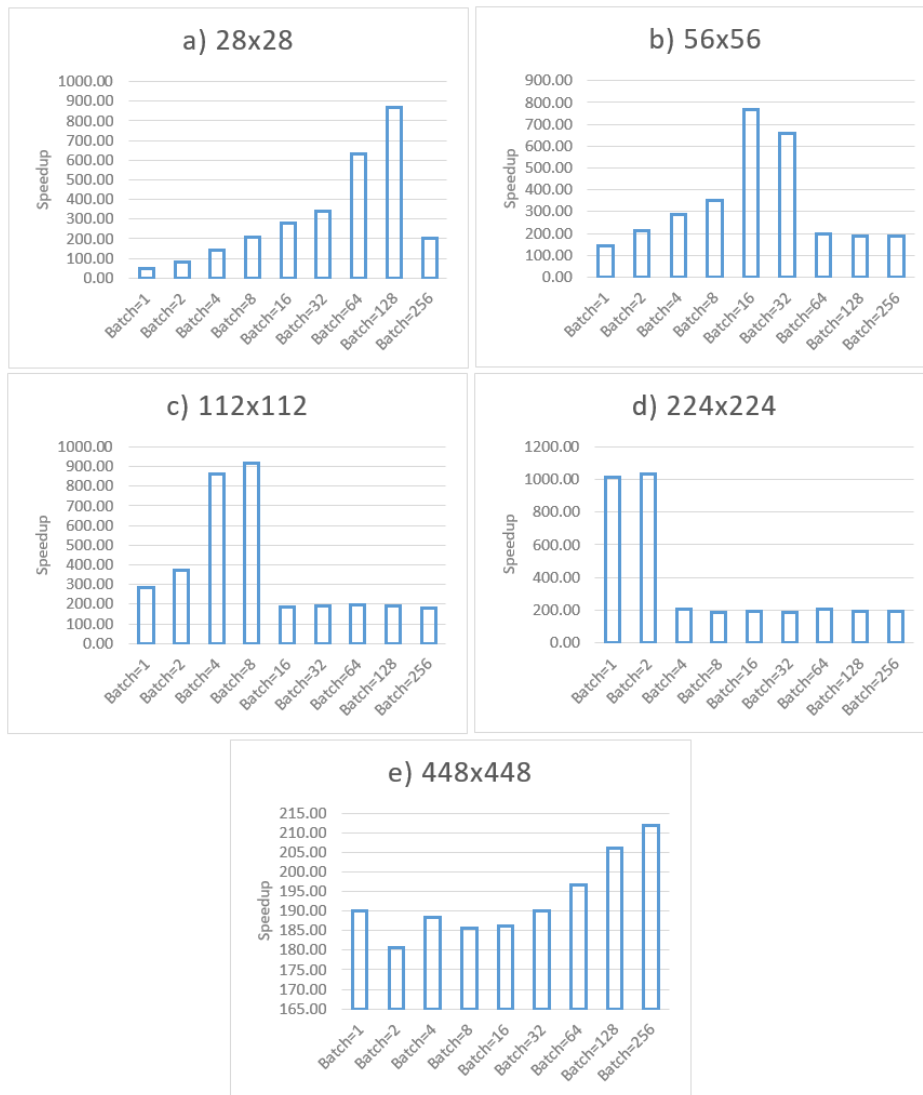


Figure 8: PyTorch speed-up over Python implementation for a) 28×28 image, b) 112×112 image c) 448×448 image

4. Discussions and Conclusions

The running times for Python scripts is usually longer than the compiled algorithms since it is an interpreted language and executes the code interpreting it line by line. Comparison with PyTorch implementation show that compiled operations provide significant accelerations

together with multicore implementation. Script implementation usually utilizes one CPU core and the increase in the computation times are approximately linear according to the number of pixels operations which are determined by image dimensions and batch size. For example the computation time for a 112×112 image where batch size is set to 16 is about 4.04 second. If the batch size is increased to 32 for the same image the computation time is increased to about 8.17 seconds which is nearly 2 times higher than the results for the batch size of 16. Similar behavior is not always observed in PyTorch results. For example the computation times for a 112×112 image are 0.0009 and 0.0014 for batch sizes 1 and 2 respectively. This can be related to the small computation times, multicore computations and variations in CPU frequency, and cache memory size. In general PyTorch library, reduces computation time to practical levels and in the case of more CPU with more cores or GPU support the results will improve especially for processing large sizes of data. Implemented, algorithm for LBP computations can be utilized as a feature extraction layer in various machine learning and deep learning algorithms for computer vision applications.

References

- Alahmadi, A., M. Hussain, H. Aboalsamh, G. Muhammad, G. Bebis, and H. Mathkour. 2017. "Passive detection of image forgery using DCT and local binary pattern". *Signal, Image and Video Processing* 11, no. 1 (january): 81-88. <https://doi.org/10.1007/s11760-016-0899-0>.
- Balandat, M., B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy. 2019. "BOTORCH: Programmable Bayesian Optimization in PyTorch". Preprint, submitted October 14, 2019. <https://arxiv.org/abs/1910.06403v1>.
- Hassaballah, M., M. A. Kenk, and I. M. El-Henawy. 2020. "Local binary pattern-based on-road vehicle detection in urban traffic scene". *Pattern Analysis and Applications* 23, no. 4 (november): 1505-21. <https://doi.org/10.1007/s10044-020-00874-9>.
- Kim, D., G. Lee, M. Lee, S. U. Kang, and D. Kim. 2020. "Normalized convolutional neural network". Preprint, submitted May 11, 2020. <https://arxiv.org/abs/2005.05274v1>.
- Ojala, T., M. Pietikäinen, and T. Mäenpää. 2002. "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, no. 7 (july): 971-87. <https://doi.org/10.1109/tpami.2002.1017623>.
- Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, et al. 2019. "PyTorch: An imperative style, high-performance deep learning library". In *Advances in Neural Information Processing Systems [33rd Annual Conference on Neural Information Processing Systems, NeurIPS 2019]*, 8026-37.
- Pereira, P. M. M., R. Fonseca-Pinto, R. P. Paiva, P. A. A. Assuncao, L. M. N. Tavora, L. A. Thomaz, and S. M. M. Faria. 2020. "Dermoscopic skin lesion image segmentation based on Local Binary Pattern Clustering: Comparative study". *Biomedical Signal Processing and Control* 59 (may): Article number 101924. <https://doi.org/10.1016/j.bspc.2020.101924>.
- Pietikäinen, M. 2010. "Local binary patterns". In *Scholarpedia*. Vol. 5, no. 3. <https://doi.org/10.4249/scholarpedia.9775>.
- Pietikäinen, M., A. Hadid, G. Zhao, and T. Ahonen. 2011. "Computer Vision Using Local Binary Patterns". In *Computer Vision Using Local Binary Patterns*. Computational Imaging and Vision, vol. 40. London: Springer. https://doi.org/10.1007/978-0-85729-748-8_14.

- Rahul, M., N. Kohli, and R. Agarwal. 2020. "Facial expression recognition using local binary pattern and modified hidden Markov model". *International Journal of Advanced Intelligence Paradigms* 17, no. 3/4. <https://doi.org/10.1504/ijaip.2020.109523>.
- Tu, B., W. Kuang, G. Zhao, D. He, Z. Liao, and W. Ma. 2019. "Hyperspectral image classification by combining local binary pattern and joint sparse representation". *International Journal of Remote Sensing* 40, no. 24 (december): 9484-500. <https://doi.org/10.1080/01431161.2019.1633699>.
- Xi, M., L. Chen, D. Polajnar, and W. Tong. 2016. "Local binary pattern network: A deep learning approach for face recognition". In *2016 IEEE International Conference on Image Processing (ICIP) - Proceedings*, 3224-28. IEEE. <https://doi.org/10.1109/ICIP.2016.7532955>.
- Xu, X., and M. Pedersoli. 2019. "A computing Kernel for network binarization on PyTorch". Preprint, submitted November 11, 2019. <https://arxiv.org/abs/1911.04477>.
- Yang, W., and S. Krishnan. 2017. "Combining temporal features by local binary pattern for acoustic scene classification". *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 25, no. 6 (june): 1315-21. <https://doi.org/10.1109/taslp.2017.2690558>.