A TENSORFLOW BASED METHOD FOR LOCAL DERIVATIVE PATTERN

Devrim AKGUN*, Software Engineering Department, Sakarya University, Turkey, dakgun@sakarya.edu.tr

Received: 24.11.2020, Accepted: 20.02.2021 Research Article
*Corresponding author
DOI: 10.22531/muglajsci.830691

Abstract

Python programming language provides a very convenient environment of implementing machine learning applications. However, programmers usually faced with a poor performance compared to compiled functions when they write script based programs that demands intense computations. TensorFlow framework provides acceleration by enabling the utilization of various computing resources such as multicore CPU and GPU unit as well as including various compiled algorithms for developing machine learning applications. In this way, algorithms developed using existing TensorFlow operations can shorten computation times by using these resources indirectly without requiring parallel programming or GPU programming. In this study, Local Derivative Pattern (LDP) analysis which is one of the efficient feature extraction algorithms for developing machine learning applications was realized using a TensorFlow based algorithm. Independent pixel based operations in LDP algorithm which requires intense computations, enable developing an efficient tensor based algorithm. The performance of the TensorFlow based algorithm has been measured by comparing it with the Python script version of the same algorithm. The results obtained for various sizes and numbers of sample images show that TensorFlow operations provide significant acceleration rates for the LDP algorithm.

Keywords: TensorFlow, Local Derivative Pattern, Feature extraction

1. Introduction

Feature extraction methods which involves most of the machine learning applications helps reduce the dimensionality of dataset while improving the learning process. LDP is of one the efficient feature extraction methods used in computer vision applications. It was first introduced in a face identification and verification study and shown to be successful on various datasets [1]. LDP checks local variations using derivative and forms a binary number according to directions. Local variations can be measured in various angles enabling detailed information about the input data. In literature there are various computer vision studies utilizing LDP for example; infrared image based fingerprint recognition [2], image annotation [3], person re-identification [4], content based medical image retrieval [5], image registration [6], facial expression recognition [7] and various face recognition approaches [8]-[10].

Özet


Anahtar Kelimeler: TensorFlow, Yerel Türev Örüntüsü, Özellik çıkarımı

Cite

Because of the pixel based multiplication and comparison operations in LDP algorithm, the computational demand of the algorithm increases as the size of the image is increased. Running the algorithm using Python may result in longer durations when compared to low level programming languages. Languages like C/C++ are closer to hardware and they are optimized during compile time to reduce running time of the program. However, Python is a scripting language where an interpreter processes input lines and convert it to machine code which adds some overhead to the execution of the program. For all that, machine learning and scientific applications are widely developed using Python because of the easy scripting languages provide. Due to its popularity one can find various resources on machine learning and numerical algorithms. TensorFlow is one of the major open source libraries in machine learning and deep learning [11], [12]. It provides various tools and makes it practical to train and test in Python.

TensorFlow also supports GPU (Graphics Processing Unit) utilization which provides significant accelerations for intense computations. Users can define their custom operations in TensorFlow as well as constructing custom layers [13]–[15]. In addition to coding from the scratch, custom operations can be defined by combining existing TensorFlow operations. In the presented study, a custom operation for LDP has been implemented using TensorFlow operations. Proposed algorithm can be used with deep learning models as intermediate layer or preprocessing function. In order to test the efficiency of the algorithm images in various dimensions such as 448×448, 224×224, 112×112, 56×56 and 28×28 were used. Also various batch sizes for each image sizes were used in experimental evaluations. Organization of the rest of the study is as follows; information about the basics of LDP was given in following section, the proposed design was explained in Section 3, numerical evaluations using TensorFlow implementation were realized in Section 4. A brief conclusions of the study were given in the last section.

2. Local Derivative Transform

LDP works similar to Local Binary Patterns (LBP) [16], but LBP only use first order derivatives with respect to neighbors within selected mask. LDP also incorporates second order derivatives based on the first order derivatives which extract more detailed features from the input image. Also the approach enables computing n-th order derivatives based on the (n-1)-th order derivatives.

Let the directional first order derivatives of input image I(P) for the directions as α=0°, α=45°, α=90° and α=135° be I′_0(P_0), I′_45(P_0), I′_90(P_0) and I′_135(P_0) respectively. First order derivatives at the p_0 location for all directions can be computed as follows,

\[
\begin{align*}
I'_0(p_0) &= I(p_0) - I(p_0) \\
I'_45(p_0) &= I(p_0) - I(p_3) \\
I'_90(p_0) &= I(p_0) - I(p_2) \\
I'_135(p_0) &= I(p_0) - I(p_1)
\end{align*}
\]

Where p_0 describes the center pixel and p_1, p_2,...,p_8 pixels within 3×3 neighborhood as shown by Figure 1.

The second-order derivatives are written based on the first order derivatives. For example, second order derivatives for p_0 can be written for α=0° as below,

\[
LDP^2_0(p_0) = \begin{cases} 
(I'(I'_0(p_0),I'_0(p_1))) \\
(I'(I'_0(p_0),I'_0(p_2))) \\
\ldots \\
(I'(I'_0(p_0),I'_0(p_8)))
\end{cases}
\]

In the same way, the equations are obtained for α=45° as below,

\[
LDP^2_{45}(p_{45}) = \begin{cases} 
(I'(I'_{45}(p_0),I'_{45}(p_1))) \\
(I'(I'_{45}(p_0),I'_{45}(p_2))) \\
\ldots \\
(I'(I'_{45}(p_0),I'_{45}(p_8)))
\end{cases}
\]

In Eq.2, f( ) contains a comparison function that produces a binary value as given by Eq. 4. It checks the derivative directions for different neighbor pixels.

\[
f(I'_a(P_0),I'_a(P_1)) = \begin{cases} 
0, & I'_a(P_0),I'_a(P_1) > 0 \\
1, & I'_a(P_0),I'_a(P_1) \leq 0
\end{cases}
\]

Computations given by Eq. 2 and Eq. 3 are repeated for other directions as follows,

\[
LDP^2(P) = \begin{cases} 
LDP^2_0(P) \\
LDP^2_{45}(P) \\
LDP^2_{90}(P) \\
LDP^2_{135}(P)
\end{cases}
\]

3. TensorFlow Implementation

TensorFlow framework provides most of the tools for developing, training and testing machine learning models. TensorFlow has APIs for several languages such as Python, Java and C++. Also it has the flexibility of defining custom functions either from scratch or as a
combination of existing functions. Because TensorFlow supports GPU computation, in the latter case GPU support is enabled for custom function without writing GPU level code. This is possible by using basic functions that have GPU support provided by TensorFlow library such as `add()`, `multiply()` and `matmul()`.

Example second order computations using Eq. 2 for $\alpha=0^\circ$, $\alpha=45^\circ$, $\alpha=90^\circ$ and $\alpha=135^\circ$ are shown by Figure 2. Operations for four angles are similar except the directions of the derivates which are computed initially. A Python realization of the algorithm for $\alpha=0^\circ$ is given by Code snippet 1. At first, Eq. 1 is applied to a pixel which is determined by for-loops and its neighborhood of $3 \times 3$ dimensions. Then Eq.2 and Eq.3 are applied to obtain second order derivatives. Extracted feature is computed by summing the resulting values after converting each digit to a decimal value.

1. #Allocate memory for output
2. L=np.zeros((rows,cols))
3. #Traverse the pixels of image
4. for i in range(2,nRows-2):
5.   for j in range(2,nCols-2):
6.     #Compute first order derivatives for $\alpha=0^\circ$ (Eq. 1)
7.     d1= I[i-1,j-1]-I[i-1,j+0]
8.     d2= I[i-1,j+0]-I[i-1,j+1]
9.     d3= I[i-1,j+1]-I[i-1,j+2]
10.    d4= I[i+0,j-1]-I[i+0,j+0]
11.    d5= I[i+0,j+1]-I[i+0,j+2]
12.    d6= I[i+1,j-1]-I[i+1,j+0]
13.    d7= I[i+1,j+0]-I[i+1,j+1]
14.    d8= I[i+1,j+1]-I[i+1,j+2]
15. #Select the pixel at the center
16.    dC= I[i,j]-I[i+0,j+0]
17. #Compute second order derivatives (Eq. 2)
18.    p1=(d1*dC)>=0)*128
19.    p2=(d2*dC)>=0)*64
20.    p3=(d3*dC)>=0)*32
21.    p4=(d4*dC)>=0)*1
22.    p5=(d5*dC)>=0)*16
23.    p6=(d6*dC)>=0)*2
24.    p7=(d7*dC)>=0)*4
25.    p8=(d8*dC)>=0)*8
26. #Compute extracted feature
27.    s=p1+p2+p3+p4+p5+p6+p7+p8
28. L[i,j]=s

Code snippet 1. Python implementation of LBP transform for $\alpha=0^\circ$.

LBP algorithm can be realized by existing TensorFlow operations since the computation of output pixels for LBP transform involves independent matrix operations. It requires various comparison, multiplication, addition and subtraction for each pixel operations and these are independent for each $3 \times 3$ mask. Therefore, the two for-loop can be eliminated in TensorFlow implementation by using tensors which are N-dimensional arrays. Code snippet 2 shows a fragment of the program using TensorFlow operations. In this code $\gamma 0$, for example is a matrix that defines the pixels at center for all masks. Similarly $\gamma 4$ defines the right neighbor of $\gamma 0$ in matrix form. Initially the first order derivatives are determined for the center pixels by using `tf.subtract()`. Before doing the comparison given by Eq.3, false and true conditions are defined. Initially, false condition is defined by `tf.zeros()` and for the following pixels it is used as previous results which in the false condition doesn’t change the result. Similarly true condition is also defined using the values according to

---

**Figure 2.** Example computations for first order LDP where $\alpha=0^\circ$, $\alpha=45^\circ$, $\alpha=90^\circ$ and $\alpha=135^\circ$.
weight of the binary digit which is 128 for $p_1$. Then, first order derivative is computed for the selected neighbor and multiplied with $p_0$ like in the Code Snippet. Following the comparison with $tf.greater_equal()$, one of the conditions are selected with $tf.where()$. After $p_1$, computations for $p_2$ are realized in the example code fragment. In this case, false condition is selected as the previous results. Other operations are the same as the previous one except the weight of the binary digit. Hence, similar operations are repeated for the other neighbor of the center pixel to obtain LDP transform for $\alpha=0^\circ$. Also this procedure is repeated for $\alpha=45^\circ$, $\alpha=90^\circ$ and $\alpha=135^\circ$ for computing all four directions. Then the obtained 8 byte results are concatenated to obtain a 32 bit feature vector and this vector is used to extract features in various applications [17], [18].

1. #compute first order derivative for center pixel
2. $dc= tf.subtract(y_0, y_4)$
3. #computation of $p_1$ in code snippet 1
4. $cond_{false}= tf.zeros(p_0.shape, tf.float32)$
5. $cond_{true}= tf.add(cond_{false}, 6. tf.constant(128.0/255.0))$
6. $d_0= tf.subtract(y_1, y_2)$
7. $d_0= tf.multiply(d_0, dc)$
8. $g= tf.greater_equal(dc, 0)$
9. $z= tf.where(g, cond_{true}, cond_{false})$
10. # computation of $p_2$ in Code Snippet 1
11. $cond_{false}=z$
12. $cond_{true}= tf.add(z, 14. tf.constant(64.0/255.0))$
15. $d_1= tf.subtract(y_2, y_3)$
16. $d_1= tf.multiply(d_1, dc)$
17. $g= tf.greater_equal(d_1, 0)$
18. $z= tf.where(g, cond_{true}, cond_{false})$
19. # computation of $p_3$ in Code Snippet 1
20. $cond_{false}=z$
21. $cond_{true}= tf.add(z, 22. tf.constant(32.0/255.0))$
23. $d_2= tf.subtract(y_3, y_4)$
24. $d_2= tf.multiply(d_2, dc)$
25. $g= tf.greater_equal(d_2, 0)$
26. $z= tf.where(g, cond_{true}, cond_{false})$

Code snippet 2. A fragment of TensorFlow implementation of LDP transform for $\alpha=0^\circ$.  

4. Experimental Results

Running times for the example cases were measured using time command from time library in Python as shown by Code Snippet 3. In that code, the features are obtained separately for four directions. In practice, these features are used together to train a machine learning model. The average of repeated measurements for 30 times to obtain example running times. Various dimensions of images and batch of images were used in tests. Experiments were carried on a hardware that contains AMD FX2700 processor and GTX1080 Nvidia GPU. Operating system was Ubuntu 18.04 where Python 3.7.9 and TensorFlow 2.3.1 were installed. Example images for obtaining performance evaluations are selected randomly from ImageNet [19] dataset. An example image and its LDP visualizations for $\alpha=0^\circ$, $\alpha=90^\circ$, $\alpha=45^\circ$ and $\alpha=135^\circ$ were given by Figure 3. In the experiments, selected images were rescaled to various dimensions such as $28\times28$, $56\times56$, $112\times112$, $224\times224$ and $448\times448$. Since the content of the image doesn’t have effect on the computation times similar experiments can be repeated using another dataset.
Running durations were first obtained for Python script for all directions. Code Snippet 1 which was given for $\alpha=0^\circ$ was realized for the other directions which are $\alpha=45^\circ$, $\alpha=90^\circ$ and $\alpha=135^\circ$. According to results given by Table 1 running times increases significantly as the images dimensions and batch size are increased. Running times are approximately related to each other linearly. For example, while running duration for $112 \times 112$ is 2.375 for the batch size 1, it increases to 4.724 when the batch size is increased to 2. The same is not true for the TensorFlow measurements where GPU and multicore CPU can be utilized. As shown by Table 2 where the results for TensorFlow implementation were given, while some measurements have linear relation when considering the batch size or image size, some measurements don’t have such relation. On the other hand, when the numerical results given by Table 1 and Table 2 are compared, it is seen that the running durations are considerably reduced. The speed-up is significantly increased as the image size and batch size are increased as shown by Figure 4.

### 5. Conclusion

LDP algorithm that extracts features from images depending on derivative operations in four directions is very useful for machine learning applications. These kinds of algorithms are usually costly to implement with Python script as the experimental results imply. TensorFlow enables users writing custom functions as the combinations of basic TensorFlow operations and these functions can be used as custom deep learning layer or preprocessing function. Since operations in TensorFlow compiled with optimization and they support multicore CPU and GPU device, running durations of custom functions are also reduced significantly when compared with the Python scripts. In order to better utilize operations, the algorithm should be expressed in terms of tensors which are $n$-dimensional arrays. In the case of LDP, since the pixels based computations are independent, the algorithm can be expressed in terms of basic operations. According to experimental running durations, custom function for LDP provides significant accelerations over Python script implementation. Also the results indicate that acceleration with TensorFlow increases as the tested image dimensions and batch size are increased which is
mainly due to the efficiency of the GPU utilization for small images deteriorates. Although the results will vary according to experimental hardware specifications and software versions, they give good insight into the TensorFlow acceleration for various dimensions of images and batch sizes.

6. References


