SVM with RBF and polynomial kernels

August 5, 2019

Contents

Here, we code SVM classifier with Gaussian Radial Basis Function (RBF) and polynomial kernels using pycuda python library. We write RBF kernel function in two ways: with and without cublas library. We code polynomial kernel function using scikit-cuda library. We then translate SVM with RBF kernel (cublas version) and with polynomial kernel to pyopencl.

1 Data

To feed the data into the SVM classifier with RBF kernel, we first normalize the MNIST dataset and then transpose it. When normalizing, centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. shuffled

Because of memory constraints, the train set size was set to 7499 and the test set size was set to 2500.

We first normalize the data:

[0]:
```
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd

df = pd.read_csv('mnist_data.csv') #file contains labels in the first column and␣
 ↪pixel values in the rest of the columns

cols = np.arange(1,785)

data = df.drop(df.columns[0],axis=1) #pixel data
```

```
label = df.drop(df.columns[cols],axis=1) #label data

sc_X = StandardScaler()

normalized_data = sc_X.fit_transform(np.array(data))
normalized_data = normalized_data.astype(np.float32)

label.to_csv('label.csv', index=False, header=False)
np.savetxt('data.csv', normalized_data, delimiter=',')
```

We then transpose the data and set class label '1' to be positive and the rest of the classes to be negative.

[0]:
```
import sklearn
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd

ref =  np.loadtxt('data.csv', delimiter=',').astype(np.float32)

label = np.loadtxt('label.csv', delimiter=',').astype(np.int32)

#set class '1' to be positive
for i in range(label.shape[0]):
    if label[i]!=1:
        label[i]=-1

#split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(ref, label, test_size=0.25,␣
 ↪shuffle = True)

#transpose the data
x_test_t = np.transpose(X_test).astype(np.float32)
x_train_t = np.transpose(X_train).astype(np.float32)
```

2 SVM classifier with pycuda

We first will define 3 functions to calculate RBF and polynomial kernels and then will test the SVM classifier with these kernels.

2.1 RBF kernel without cublas

We now write SVM classifier in python using pycuda library. We calculate the distance between the feature vectors without using cublas library. In the next section, we will use cublas library to do this.

The RBF kernel computation consists of two parts: the first part is calculating the squared Euclidean distance between feature vectors and then use it when calculating the RBF kernel matrix.

So we first write a kernel to calculate the distance matrix between the feature vectors using pycuda library:

[0]:
```
import pycuda.autoinit
import pycuda.driver as drv
import numpy as np

from pycuda.compiler import SourceModule


compute_distances_mod = SourceModule("""
__global__ void compute_distances(float * ref,
                                  int     ref_width,
                                  int     ref_pitch,
                                  float * query,
                                  int     query_width,
                                  int     query_pitch,
                                  int     height,
                                  float * dist) {
    const int BLOCK_DIM = 16;
    // Declaration of the shared memory arrays As and Bs used to store the␣
 ↪sub-matrix of A and B
    __shared__ float shared_A[BLOCK_DIM][BLOCK_DIM];
    __shared__ float shared_B[BLOCK_DIM][BLOCK_DIM];

    // Sub-matrix of A (begin, step, end) and Sub-matrix of B (begin, step)
    __shared__ int begin_A;
    __shared__ int begin_B;
    __shared__ int step_A;
    __shared__ int step_B;
    __shared__ int end_A;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;


    // Initializarion of the SSD for the current thread
    float ssd = 0.f;

    // Loop parameters
    begin_A = BLOCK_DIM * blockIdx.y;
    begin_B = BLOCK_DIM * blockIdx.x;
    step_A  = BLOCK_DIM * ref_pitch;
    step_B  = BLOCK_DIM * query_pitch;
    end_A   = begin_A + (height-1) * ref_pitch;

    // Conditions
```

```
    int cond0 = (begin_A + tx < ref_width); // used to write in shared memory
    int cond1 = (begin_B + tx < query_width); // used to write in shared memory␣
↪& to computations and to write in output array
    int cond2 = (begin_A + ty < ref_width); // used to computations and to write␣
↪in output matrix

    // Loop over all the sub-matrices of A and B required to compute the block␣
↪sub-matrix
    for (int a = begin_A, b = begin_B; a <= end_A; a += step_A, b += step_B) {

        // Load the matrices from device memory to shared memory; each thread␣
↪loads one element of each matrix
        if (a/ref_pitch + ty < height) {
            shared_A[ty][tx] = (cond0)? ref[a + ref_pitch * ty + tx] : 0;
            shared_B[ty][tx] = (cond1)? query[b + query_pitch * ty + tx] : 0;
        }
        else {
            shared_A[ty][tx] = 0;
            shared_B[ty][tx] = 0;
        }

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Compute the difference between the two matrixes; each thread computes␣
↪one element of the block sub-matrix
        if (cond2 && cond1) {
            for (int k = 0; k < BLOCK_DIM; ++k){
                float tmp = shared_A[k][ty] - shared_B[k][tx];
                ssd += tmp*tmp;
            }
        }

        // Synchronize to make sure that the preceeding computation is done␣
↪before loading two new sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory; each thread writes one␣
↪element
    if (cond2 && cond1) {
        dist[ (begin_A + ty) * query_pitch + begin_B + tx ] = ssd;
    }
}

""")
```

And then write a CUDA kernel to calculate the RBF kernel matrix:

[0]:
```
mod = SourceModule("""
__global__ void rbf_kernel(float * data,
                           int     query_nb,
                           int     ref_nb,
                           float sigma)

{

int idx = threadIdx.x+blockIdx.x*blockDim.x;

if (idx<query_nb*ref_nb){
data[idx]  /= -2.0*sigma*sigma;
data[idx] = exp(data[idx]);
}
}
""")
```

We now define a function to calculate the rbf kernel matrix, which will then be fed into the training function of our SVM classifier. Note that the data passed to the rbf function has nxm shape where n is the number of features and m is the number of examples (i.e. the input data is transposed).

[0]:
```
def rbf_kernel(ref, query, sigma):
    #data parameters
    ref_nb = np.int32(ref.shape[1]) #the number of reference points
    query_nb = np.int32(query.shape[1]) #the number of query points
    dim = np.int32(query.shape[0]) #dimension of data = the number of features
    knn_dist = np.zeros([ref_nb, query_nb]).astype(np.float32)

    BLOCK_DIM = 16 #dimension of blocks

    #Allocate memory on device
    dist_dev = drv.mem_alloc(knn_dist.nbytes)

    #Copy data from the host to the device
    drv.memcpy_htod(dist_dev, knn_dist)

    #Deduce pitch values (not in bytes)
    ref_pitch = np.int32(ref.strides[0]/ref.dtype.itemsize)
    query_pitch = np.int32(query.strides[0]/query.dtype.itemsize)
    dist_pitch = np.int32(knn_dist.strides[0]/knn_dist.dtype.itemsize)


    #Check pitch values
    if (query_pitch != dist_pitch):
        print('ERROR: Invalid pitch value')
```

```
    grid_x = query_nb / BLOCK_DIM
    grid_y = ref_nb / BLOCK_DIM
    if (query_nb % BLOCK_DIM != 0):
        grid_x += 1
    if (ref_nb   % BLOCK_DIM != 0):
        grid_y += 1


    #Compute the squared Euclidean distances
    compute_distances = compute_distances_mod.get_function("compute_distances")
    compute_distances(ref, ref_nb, ref_pitch, query, query_nb, query_pitch, dim,␣
↪dist_dev, block=(16,16,1), grid = (int(grid_x), int(grid_y),1))

    grid_x = query_nb*ref_nb / 32

    if (query_nb*ref_nb % 32 != 0):
        grid_x += 1

    rbf = mod.get_function("rbf_kernel")
    rbf(dist_dev, query_nb, ref_nb, sigma, block=(32,1,1),␣
↪grid=(int(grid_x),1,1))

    return dist_dev
```

2.2 RBF kernel with cublas

In this section, we calculate the distances between the feature vectors using cublas library (the distances are calculated in the same way as in the kNN classifier I sent earlier). The rest is the same as in the previous section.

We first define CUDA kernels for distance matrix calculation:

[0]:
```
import pycuda.autoinit
import pycuda.driver as drv
import numpy as np
from pycuda.compiler import SourceModule
import pycuda.gpuarray as gpuarray
import random
import math
import time
import skcuda.cublas as cublas

#CUDA KERNELS
"""
 * Computes the squared norm of each column of the input array.
 *
 * @param array   input array
```

```
 * @param width   number of columns of `array` = number of points
 * @param pitch   pitch of `array` in number of columns
 * @param height  number of rows of `array` = dimension of the points
 * @param norm    output array containing the squared norm values
"""

compute_squared_norm_mod = SourceModule("""
    __global__ void compute_squared_norm(float * array, int width, int pitch,␣
 ↪int height, float * norm){
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex<width){
        float sum = 0.f;
        for (int i=0; i<height; i++){
            float val = array[i*pitch+xIndex];
            sum += val*val;
        }
        norm[xIndex] = sum;
    }
}
""")




"""
 * Add the reference points norm (column vector) to each colum of the input␣
 ↪array.
 *
 * @param array   input array
 * @param width   number of columns of `array` = number of points
 * @param pitch   pitch of `array` in number of columns
 * @param height  number of rows of `array` = dimension of the points
 * @param norm    reference points norm stored as a column vector
"""

add_reference_points_norm_mod = SourceModule("""
__global__ void add_reference_points_norm(float * array, int width, int pitch,␣
 ↪int height, float * norm){
    unsigned int tx = threadIdx.x;
    unsigned int ty = threadIdx.y;
    unsigned int xIndex = blockIdx.x * blockDim.x + tx;
    unsigned int yIndex = blockIdx.y * blockDim.y + ty;
    __shared__ float shared_vec[16];
    if (tx==0 && yIndex<height)
        shared_vec[ty] = norm[yIndex];
    __syncthreads();
    if (xIndex<width && yIndex<height)
```

```
        array[yIndex*pitch+xIndex] += shared_vec[ty];
}
""")

"""/**
 * Adds the query points norm (row vector) to the input
 *
 * @param array   input array
 * @param width   number of columns of `array` = number of points
 * @param pitch   pitch of `array` in number of columns
 * @param k       number of neighbors to consider
 * @param norm     query points norm stored as a row vector
"""

add_query_points_norm_mod = SourceModule("""
__global__ void add_query_points_norm(float * array, int width, int pitch, int␣
 ↪k, float * norm){
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
    if (xIndex<width && yIndex<k)
        array[yIndex*pitch + xIndex] = array[yIndex*pitch + xIndex] +␣
 ↪norm[xIndex];
}
""")
```

And then we put them into the function which calculates the RBF matrix:

[0]:
```
def cublas_rbf_kernel(ref, query, sigma):

    #data parameters
    ref_nb = np.int32(ref.shape[1]) #the number of reference points
    query_nb = np.int32(query.shape[1]) #the number of query points
    dim = np.int32(query.shape[0]) #dimension of data = the number of features
    knn_dist = np.zeros([ref_nb, query_nb]).astype(np.float32)

    BLOCK_DIM = 16 #dimension of blocks

    #Allocate memory on device
    dist_dev = drv.mem_alloc(knn_dist.nbytes)

    #Copy data from the host to the device
    drv.memcpy_htod(dist_dev, knn_dist)

    #Deduce pitch values (not in bytes)
    ref_pitch = np.int32(ref.strides[0]/ref.dtype.itemsize)
    query_pitch = np.int32(query.strides[0]/query.dtype.itemsize)
    dist_pitch = np.int32(knn_dist.strides[0]/knn_dist.dtype.itemsize)
```

```
    #Check pitch values
    if (query_pitch != dist_pitch):
        print('ERROR: Invalid pitch value')
    #Initialize CUBLAS
    context = cublas.cublasCreate()


    BLOCK_DIM = 16

    ref_norm_dev =  drv.mem_alloc(ref.nbytes)
    query_norm_dev = drv.mem_alloc(query.nbytes)

    #Copy data from host to device
    drv.memcpy_htod(dist_dev, knn_dist)


    #Compute the squared norm of the reference points
    compute_squared_norm = compute_squared_norm_mod.
↪get_function("compute_squared_norm")

    grid_x = ref_nb / 256
    if (ref_nb % 256 != 0):
        grid_x += 1

    compute_squared_norm(ref.gpudata, ref_nb, ref_pitch, dim, ref_norm_dev,
        block=(256,1,1), grid=(int(grid_x),1,1))


    # Compute the squared norm of the query points
    compute_squared_norm = compute_squared_norm_mod.
↪get_function("compute_squared_norm")

    grid_x = query_nb / 256
    if (query_nb % 256 != 0):
        grid_x += 1

    compute_squared_norm(query.gpudata, query_nb, query_pitch, dim,␣
↪query_norm_dev,
        block=(256,1,1), grid=(int(grid_x),1,1))


    # Computation of query*transpose(reference)
    cublas.cublasSgemm(context, 'n', 't', query_pitch, ref_pitch, dim, np.
↪float32(-2.0), query.gpudata, query_pitch, ref.gpudata, ref_pitch, np.
↪float32(0.0), dist_dev, query_pitch)
```

```
    cublas.cublasDestroy(context)

    # Add reference points norm
    add_reference_points_norm = add_reference_points_norm_mod.
↪get_function("add_reference_points_norm")

    grid_x = query_nb / 16
    grid_y = ref_nb / 16

    if (query_nb % 16 != 0):
        grid_x += 1

    if (ref_nb % 16 != 0):
        grid_y += 1

    add_reference_points_norm(dist_dev, query_nb, dist_pitch, ref_nb,␣
↪ref_norm_dev,
        block=(16,16,1), grid=(int(grid_x),int(grid_y),1))


    # Add query norm and compute the square root of the of the k first elements
    add_query_points_norm = add_query_points_norm_mod.
↪get_function("add_query_points_norm")

    grid_x = query_nb / 16
    grid_y = ref_nb / 16
    if (query_nb % 16 != 0):
        grid_x += 1
    if (ref_nb % 16 != 0):
        grid_y += 1

    add_query_points_norm(dist_dev, query_nb, dist_pitch, ref_nb, query_norm_dev,
        block=(16, 16, 1), grid=(int(grid_x),int(grid_y),1))

    grid_x = query_nb*ref_nb / 32

    if (query_nb*ref_nb % 32 != 0):
        grid_x += 1

    rbf = mod.get_function("rbf_kernel")
    rbf(dist_dev, query_nb, ref_nb, sigma, block=(32,1,1),␣
↪grid=(int(grid_x),1,1))

    return dist_dev
```

calculating rbf kernel matrix for 21k training examples took 4.53 sec

2.3 Polynomial kernel

Next, we define our polynomial kernel.

We first write a CUDA kernel to raise the elements of an array to a given power:

[0]:
```
power_mod = SourceModule("""
__global__ void power(float * array, int order, int num){
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    if (i<num){
    array[i]= pow(array[i], order);
}
}


""")
```

Then, we define a function that first calculates the dot product between the first matrix and the transpose of the second one, multiples it by gamma and adds a coefficient. The result is then passed to a CUDA kernel that raises the elements of the input matrix to a given power.

[0]:
```
from pycuda.compiler import SourceModule
import pycuda.gpuarray as gpuarray
import math
import pycuda.autoinit
import pycuda.driver as drv
import numpy as np
import skcuda

culinalg.init()

def polynomial_kernel(x,y, gamma, coeff, order):
    temp_gpu = gamma*skcuda.linalg.dot(x, skcuda.linalg.transpose(y)) + coeff

    p = power_mod.get_function("power")
    x,y = temp_gpu.shape
    p(temp_gpu, np.int32(order), np.int32(x*y), block=(int(math.ceil(x*y/1024.
 ↪0)),1,1), grid=(1024,1,1))
    return temp_gpu
```

3 Train & Test

Next, we write a function to train the classifier:

[0]:
```
#FIT FUNCTION
def svm_fit(data, label, feature_size, batch_size, epochs, c, train_size):

    #allocate memory
    w_np = np.zeros([1,feature_size]).astype(np.float32)
    w = drv.mem_alloc(w_np.nbytes)
```

```
    drv.memcpy_htod(w, w_np)

    arr = np.ones([batch_size,feature_size]).astype(np.float32)
    xi = drv.mem_alloc(arr.nbytes)

    yi_xi_np = np.ones([1,feature_size]).astype(np.float32)
    yi_xi = drv.mem_alloc(yi_xi_np.nbytes)

    next_w_np = np.zeros([1,feature_size]).astype(np.float32)
    next_w = drv.mem_alloc(next_w_np.nbytes)



    #training
    for t in range(1, epochs) :

        drv.memcpy_htod(next_w, next_w_np)

        nt = 1/(c*t)

        idx = random.randint(1,100000) % (train_size - batch_size)

        copy_batch = copy_batch_mod.get_function("copy_batch")
        copy_batch(data, xi, np.int32(idx), feature_size, batch_size,␣
↪block=(int(math.ceil(batch_size*feature_size/63000.0)),1,1), grid=(63000,1,1))
        drv.memcpy_dtoh(arr, xi)


        select_samples = select_samples_mod.get_function("select_samples")
        select_samples(xi, w, feature_size, np.int32(idx), label.gpudata,␣
↪batch_size, block=(int(math.ceil(batch_size/512.0)),1,1), grid=(512,1,1))
        drv.memcpy_dtoh(arr, xi)


        reduce_by_samples = reduce_by_samples_mod.
↪get_function("reduce_by_samples")
        reduce_by_samples(yi_xi, xi, batch_size, feature_size, np.int32(idx),␣
↪label.gpudata, block=(int(math.ceil(feature_size/512.0)),1,1), grid=(512,1,1))
        drv.memcpy_dtoh(yi_xi_np, yi_xi)


        update_w = update_w_mod.get_function("update_w")
        update_w(yi_xi, w, next_w, np.float32(nt), c, batch_size, feature_size,␣
↪block=(int(math.ceil(feature_size/512.0)),1,1), grid = (512,1,1))


        copy_array = copy_array_mod.get_function("copy_array")
```

```
        copy_array(w, next_w, feature_size, block=(int(math.ceil(feature_size/
↪512.0)),1,1), grid = (512,1,1))


    return w
```

We now write a function to make predictions:

[0]:
```
def predict(data,feature_size, w, num):

    y_pred = np.zeros([1, num]).astype(np.int32)
    d_predicted_labels = drv.mem_alloc(y_pred.nbytes)

    predict_array = predict_array_mod.get_function("predict_array")
    predict_array(data, w, d_predicted_labels, feature_size, num, block = (math.
↪ceil(num/1875.0),1,1), grid = (1875,1,1))

    drv.memcpy_dtoh(y_pred, d_predicted_labels)

    return y_pred
```

And a function to calculate accuracy:

[0]:
```
def accuracy (label, pred_label):
    corr = 0
    for i in range(pred_label.shape[1]):
        if (label[i] == pred_label[0][i]):
            corr +=1
    return corr/label.shape[0]
```

The next function loads the training and testing data:

[0]:
```
def load_data():
    ref =  np.loadtxt('train.csv', delimiter=',').astype(np.float32)
    label = np.loadtxt('train_label.csv').astype(np.int32)
    test =  np.loadtxt('test.csv', delimiter=',').astype(np.float32)
    test_label = np.loadtxt('test_label.csv').astype(np.int32)
    return ref, label, test, test_label
```

Finally, we put everything together to train and test the classifier:

[0]:
```
def main():

    #load data
    train_data, train_label, test_data, test_label = load_data()


    #Parameters
    c = np.float32(0.0001) #the penalty parameter
    epochs = np.int32(500)
    num_iterations = np.int32(5)
    positive_class = np.int32(1)
```

```
   sigma = np.float32(5.0)
   batch_size = np.int32(200)
   feature_size = np.int32(train_data.shape[1]) #the number of distances to the␣
↪reference points
   test_size = np.int32(test_data.shape[1]) #the number of test examples
   train_size = feature_size #the number of train examples, which is obviously␣
↪equal to the feature_size since feature size is the number of distances to the␣
↪training points
   print ("Started training SVM: ")

   start = time.time()

   #calculate rbf matrix without cublas
   kernel_matrix = rbf_kernel(gpuarray.to_gpu(train_data), gpuarray.
↪to_gpu(train_data), sigma)

   #calculate rbf matrix with cublas
   #kernel_matrix = rbf_kernel_cublas(gpuarray.to_gpu(train_data), gpuarray.
↪to_gpu(train_data), sigma)

   #calculate polynomial kernel matrix
   #kernel_matrix = polynomial_kernel(gpuarray.to_gpu(train_data), gpuarray.
↪to_gpu(train_data), np.int32(1), np.int32(1), np.int32(3)):

   end = time.time()
   delta = end-start
   print('Computing rbf kernel took: %.2f sec' % delta)

   mean_accuracy = [] #list to store accuracy values

   for i in range(num_iterations):
       start = time.time()
       w = svm_fit(kernel_matrix, gpuarray.to_gpu(train_label), feature_size,␣
↪batch_size, epochs, c, train_size)
       end = time.time()
       delta = end-start
       print('Training took: %.2f sec' % delta)

       #calculte rbf kernel matrix
       test_kernel_matrix = rbf_kernel(gpuarray.to_gpu(test_data), gpuarray.
↪to_gpu(train_data), sigma)

       #calculate rbf matrix with cublas
       #test_kernel_matrix = rbf_kernel_cublas(gpuarray.to_gpu(test_data),␣
↪gpuarray.to_gpu(train_data), sigma)
```

```
        #calculate polynomial kernel matrix
        #test_kernel_matrix = polynomial_kernel(gpuarray.to_gpu(test_data),␣
↪gpuarray.to_gpu(train_data), np.int32(1), np.int32(1), np.int32(3)):

        #make predictions
        pred_label = predict(test_kernel_matrix, feature_size, w, np.
↪int32(test_data.shape[1]))
        acc = accuracy(test_label, pred_label)
        mean_accuracy.append(acc)
        print('Accuracy in the given iteration: %.4f' % acc)
    print('Mean accuracy: %.4f' % np.mean(np.array(mean_accuracy)))

    return 0



if __name__ == "__main__":
    main()
```

4 Results

Note that for SVM classifier with RBF kernel, the classification results are highly dependent on the choice of sigma parameter.

With pycuda, computing rbf kernel with cublas took 0.87 sec.

With pycuda, computing rbf kernel without cublas took 0.85 sec.

With pyopencl, computing rbf kernel took 0.58 sec.

Mean accuracy with sigma = 3.0: 97.3%

Mean accuracy with sigma = 5.0: 98.8%.

For comparison, for the same number of examples, computing polynomial kernel with pycuda took 0.16 sec.