Report: Image Segmentation

Computer Vision - Assignment 2

Ian SCHOLL (ischoll@student.ethz.ch)

1 Mean-Shift Algorithm (Total: 40 pts)

1.1 Implement the distance Function (12 pts)

The distance function I implemented first calculates the difference between the sample point x and all other sample points (including itself, radius = +∞) in X in a vectorized manner. Afterwards, I calculate the norm of each of these difference vectors again in a vectorized manner, as looping through each row of X to compute the norms is computationally way too expensive and slow. This results in a vector *dist*, with the euclidean distances between the sample point and each of the other pixels in the CIELAB space for the scaled image:

```
def distance(x, X):
    dist = torch.norm(X-x, dim=1)
    return dist
```

1.2 Implement the gaussian Function (12 pts)

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right)$$

We can now use the obtained *dist* vector to compute the width using a Gaussian kernel as for example noted in https://en.wikipedia.org/wiki/Mean_shift (01.11.2021). This results in the following – again vectorized - calculation, giving the vector *weights* as output:

```
def gaussian(dist, bandwidth):
    weights = torch.exp(-(dist**2)/(2*(bandwidth**2)))
    return weights
```

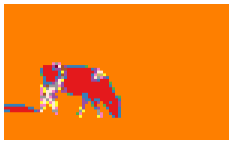
1.3 Implement the update point Function (11 pts)

The *weight* vector first has to be reshaped vertical, to then multiply element-wise with the rows of X, in order to obtain a 'weighted X'. The value can then be updated with the total-weight-normalized sum of the rows (pixels) of this 'weighted X':

```
def update_point(weight, X):
    weight = weight.reshape(len(weight), 1)
    weighted_X = torch.mul(weight, X)
    update_value = torch.sum(weighted_X, 0)/torch.sum(weight, 0)
    return update_value
```

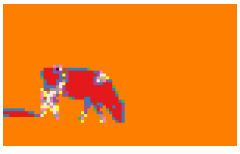
1.4 Accelerating the Naive Implementation (5 pts)

Running this setup with the given for-loop-based *meanshift_step* on the CPU of my laptop results in the expected output which looks exactly like the provided *'sample-result.png'*, however the run-time necessary is around 45 seconds:



```
(CV21_Image_Segmentation) ischoll@ischoll-ThinkPad-X1:~/ETH/MA1/CV/Assignments
/Assignment2/mean-shift_cow_student/mean-shift_cow$ python mean-shift.py
Elapsed time for mean-shift: 44.94065809249878
```

To make everything more efficient, the data can also be processed as batches. In that case, slightly adapted methods *distance_batch*, *update_point_batch* and *meanshift_step_batch* are used as seen below. Instead of calculating everything point by point, I use torch's method *cdist* in *distance_batch* and matrix multiplication to replace element-wise multiplication in the *update_point_batch*, in order to process several points at the same time. Finally, I divide the matrix *X* into batches with *batch_size* rows in the *meanshift_step_batch* method. Each batch then runs through the series *distance_batch-gaussian-update_point_batch* and updates a part of the rows of the original *X*. After all batches have been looped through, a complete updated *X* is ready for the next epoch. I have found some of the best results with a *batch_size* of 55. When now running this new approach on the same CPU of the same laptop, the expected output still looks the exact same, but the run-time has significantly reduced to just below 1sec:



```
(CV21_Image_Segmentation) ischoll@ischoll-ThinkPad-X1:~/ETH/MA1/CV/Assignments
/Assignment2/mean-shift_cow_student/mean-shift_cow$ python mean-shift.py
Elapsed time for mean-shift: 0.882033109664917
```

```
def distance_batch(x, X):
    dist = torch.cdist(x, X, p=2)
    return dist
```

```
def update_point_batch(weight, X):
    weighted_X = torch.matmul(weight, X)
    weight_sum = torch.sum(weight, 1)
    weight_sum = weight_sum.reshape(len(weight_sum), 1)
    update_value = weighted_X/weight_sum
    return update_value
```

```
def meanshift_step_batch(X, bandwidth=2.5):
    X_ = X.clone()
    batch_size = 55
    dist = torch.zeros(batch_size, X.shape[0]).double()
    for i in range(0, len(X), batch_size):
        x = X[i:i+batch_size]
        dist = distance_batch(x, X)
        weight = gaussian(dist, bandwidth)
        X_[i:i+batch_size] = update_point_batch(weight, X)
    return X_
```

Therefore, we can observe that batchifying the input can help to reduce the run-time by a factor of ~50 in my case. This shows that a fine-tuned approach, where we process a reasonable amount of data simultaneously, very often can be worth it in order to complete a task more efficiently.