Multidimensional Spectral Hashing: Hashing with the Kernel Trick

Yair Weiss, Rob Fergus, and Antonio Torralba

School of Computer Science, Hebrew University
Dept. of Computer Science, Courant Institute, New York University
CSAIL, MIT

yweiss@cs.huji.ac.il,fergus@cs.nyu.edu,torralba@csail.mit.edu

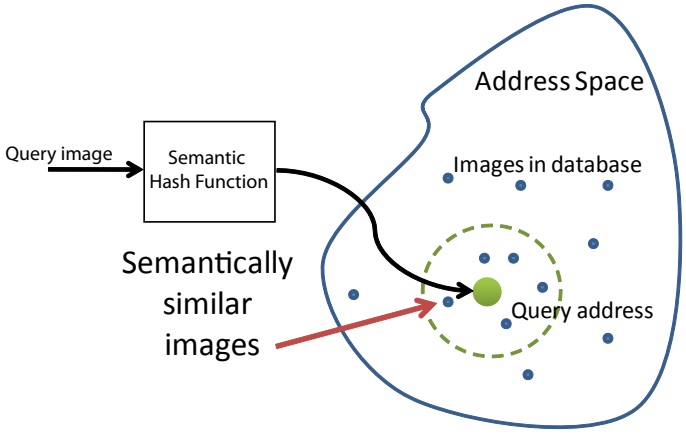

Fig. 1. The basic idea of semantic hashing. In addition to the bucket of the query, we also retrieve items in neighboring buckets. In this short note we show how to extend this idea to multidimensional spectral hashing.

Figure 1 illustrates the basic idea of semantic hashing. Given a query, we retrieve not just items with identical codes but also items with "similar codes". In the classic version of semantic hashing, the set of "similar codes" is defined to be all codes withing Hamming distance one of the query, so retrieval can be accomplished with a total of $K+1$ lookups, where K is the number of bits in the code. In MDSH the set of "similar codes" needs to be defined more carefully.

In MDSH, each datapoint i is represented by a binary vector of length K which we denote by $y_i \in \{-1,1\}^K$. During learning, the algorithm also learns a vector of length K weights $\lambda \in R^K$. The Hamming affinity between point i and j is given by:

$$H(i,j) = -1 + \prod_d \left(1 + \sum_{k \in d} \lambda_k y_i(k) y_j(k)\right) \tag{1}$$

Equation 1 can be shown to be an efficient calculation of a weighted Hamming distance between two much longer bit vectors.

In order to perform semantic hashing with equation 1 we need to quickly find "similar codes", i.e. for a given query whose code is given by y_i to efficiently find other binary codes y_j whose affinity (given by equation 1) is large. Since the affinity depends on the weights, it does not make sense to choose all codes that differ only by one bit.

The key property of equation 1 that we can use is that the affinity depends only on a vector δ_{ij} which is related to the Hamming distance between y_i and y_j:

$$\delta_{ij}(k) = \begin{cases} +1 \ , y_i(k) = y_j(k) \\ -1 \ , y_i(k) \neq y_j(k) \end{cases} \tag{2}$$

Or equivalently: $\delta_{ij}(k) = y_i(k) y_j(k)$. Using this notation, we can rewrite equation 1 as:

$$H(i,j) = -1 + \prod_d \left(1 + \sum_{k \in d} \lambda_k \delta_{ij}(k)\right) \tag{3}$$

As an example, the affinity between $y_1 = (1,1,1,1)$ and $y_2 = (-1,1,1,1)$ will be the same as the affinity between $y_3 = (1,1,-1,-1)$ and $y_4(-1,1,-1,-1)$ since both cases the vectors differ only in the first location and $\delta_{12} = \delta_{34} = (-1,1,1,1)$.

This means that we can precompute a set of "directions" $\{\delta_l\}_{l=1}^K$ and then define the set of neighbors of query y_i by $y_l(k) = y_i(k)\delta_l(k)$. The Hamming affinity of y_i to y_l will (as we noted above) depend only on the elementwise product $\delta_{il}(k) = y_i(k) y_l(k) = y_i^2 \delta_l(k) = \delta_l(k)$ and so will not depend on the query point y_i and only on the direction δ_l. So we can precompute these directions once during training, and use them for all queries.

In our current implementation we try (during learning) all codes that are Hamming distance one or two from the all-ones codeword. We then choose the $K+1$ directions that have the highest affinity to the all ones codeword. These directions are subsequently used for all queries. Thus we can still do retrieval with a total of $K+1$ lookups.