Learning "Forgiving" Hash Functions: Algorithms and Large Scale Tests

Shumeet Baluja & Michele Covell
Google, Inc.
1600 Amphitheatre Parkway, Mountain View, CA. 94043
{shumeet,covell}@google.com

Abstract

The problem of efficiently finding similar items in a large corpus of high-dimensional data points arises in many real-world tasks, such as music, image, and video retrieval. Beyond the scaling difficulties that arise with lookups in large data sets, the complexity in these domains is exacerbated by an imprecise definition of similarity. In this paper, we describe a method to learn a similarity function from only weakly labeled positive examples. Once learned, this similarity function is used as the basis of a hash function to severely constrain the number of points considered for each lookup. Tested on a large real-world audio dataset, only a tiny fraction of the points (~0.27%) are ever considered for each lookup. To increase efficiency, no comparisons in the original high-dimensional space of points are required. The performance far surpasses, in terms of both efficiency and accuracy, a state-of-the-art Locality-Sensitive-Hashing based technique for the same problem and data set.

1 Introduction

This work is motivated by the need to retrieve similar audio, image, and video data from extensive corpora of data. The large number of elements in the corpora, the high dimensionality of the points, and the imprecise nature of "similar" make this task challenging in real world systems.

Throughout this paper, we ground our discussion in a real-world task: given an extremely short (~1.4 second) audio "snippet" sampled from anywhere within a large database of songs, determine from which song that audio snippet came. We will use our system for retrieval from a database of 6,500 songs with 200 snippets extracted from each song, resulting in 1,300,000 snippets (Figure 1). The short duration of the snippets makes this particularly challenging.

The task of distortion-robust fingerprinting of music has been widely examined. Many published systems created to this point attempt to match much longer song snippets, some report results on smaller datasets, and others use prior probabilities to scale their systems [Ke et *al*., 2005; Haitsma & Kalker, 2002; Burges et *al.,* 2003; Shazam, 2005]. We assume a uniform prior and match extremely small snippets. While our system can easily be incorporated into those designed for longer snippet recognition, by testing on short snippets we highlight the fundamental retrieval issues that are often otherwise masked through the use of extra temporal coherency constraints made possible with longer snippets. Further, we will demonstrate improved performance on the retrieval of both short and long snippets.

1.1 Background

Generally stated, we need to learn how to retrieve examples from a database that are similar to a probe example in a manner that is both efficient and compact. One way to do this is to learn a distance metric that (ideally) forces the smallest distance between points that are known to be dissimilar to be larger than the largest distance between points that are known to be similar [Hastie & Tibrishani, 1996; Shental *et al*., 2002; Bar-Hillel *et al*., 2002; Tsang *et al*., 2005]. These methods can be used with *k*-nearest neighbors (*k*nn) approaches. *k*nn approaches are well suited to our task: They work with many classes and are able to dynamically add new classes without retraining. Approximate *k*nn is efficiently implemented through Locality-Sensitive Hashing (LSH) [Gionis *et al*., 1999]. LSH and other hash functions are sublinear in the number of elements examined compared to the size of the database.

LSH works for points with feature vectors that can be compared in a Euclidean space. The general idea is to partition the feature vectors into l subvectors and to hash each point into l separate hash tables, each hash table using one of the subvectors as input to the hash function. Candidate

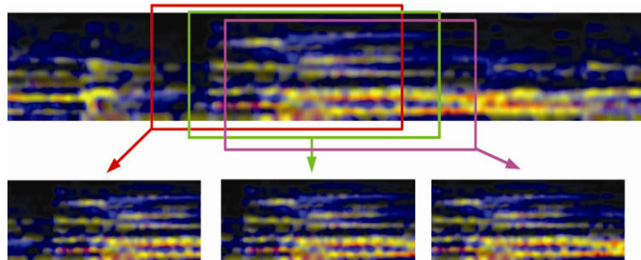

Figure 1: A typical spectrogram and extracted snippets; note the overlap. The task: given any snippet, find others from the same song.

neighbors can then be efficiently retrieved by partitioning the probe feature vector and collecting the entries in the corresponding hash bins. The final list of potential neighbors can be created by vote counting, with each hash casting votes for the entries of its indexed bin, and retaining the candidates that receive some minimum number of votes, v_t. If $v_t = 1$, this takes the union of the candidate lists. If $v_t = l$, this takes the intersection.

This idea has been extended by [Shakhnarovich, 2006] to Parameter-Sensitive Hashing (PSH). To remove the Euclidean requirement, PSH uses paired examples, both positive (similar) and negative (dissimilar) to learn the l hash functions that best cluster the two halves of each positive example while best separating the two halves of each negative example. This approach is closest to the one presented here. One difference is the choice of learning constraints. [Shakhnarovich, 2006] used positive examples (similar pairs) and explicitly provided negative examples (dissimilar pairs) to induce the learning algorithm to find a hash function. The unstated assumption is that any hash function that adequately learns the negative examples will provide (nearly) equal occupancy on the various hash bins so that the validation step does not then process unpredictably large numbers of potential neighbors. In our work, the learning function is created using only weakly labeled positive examples (similar pairs) coupled with a forced constraint towards maximum entropy (nearly uniform occupancy). We *do not* explicitly create negative examples (dissimilar pairs), but instead rely on our maximum-entropy constraint to provide that separation. As will be shown, our learning constraints have the advantage of allowing for similarities between only nominally distinct samples, without requiring the learning structure to attempt to discover minor (or nonexistent) differences in these close examples.

The remainder of this paper is organized as follows. The next section will describe the properties we need for learning a hash function, the training procedure and the analysis. Section 3 shows how the system can be scaled to handle large amounts of data. Section 4 demonstrates this concretely on a real-world audio-retrieval task. Section 5 closes the paper with conclusions and future work.

2 Learning Hash Functions

All deterministic hash functions map the same point to the same bin. Our goal is to create a hash function that also groups "similar" points in the same bin, where similar is defined by the task. We call this a *forgiving hash function* in that it forgives differences that are small with respect to the implicit distance function being calculated.

To learn a forgiving hash function for this task, we need to satisfy several objectives: (1) The hash function must be able to work without explicitly defining a distance metric. (2) The hash function must learn with only weakly labeled examples; in our task, we have indications of what points are similar (the song label), but we do not have information of which *parts* of songs sound similar to other parts. (3) The hash function must be able to generalize beyond the examples given; we will not be able to train it on samples from all the songs with which we expect to use it. Effectively, this means that the learned function must maintain entropy in the hashes for even new samples. Whether or not the songs have been seen before, they must be well distributed across the hash bins. If entropy is not maintained, points will be hashed to a small number of bins, thereby rendering the hash ineffective.

The requirement to explicitly control entropy throughout training is a primary concern. In the next section, we demonstrate the use of a neural network with multiple outputs to learn a hash function. Entropy is explicitly controlled by carefully constructing the outputs of the training examples. Other learning methods could also have been used; however, neural networks [Hertz et *al.,* 1991; Pollack, 1990] are suited to this task since, as we will show, controlling the entropy of each output and between outputs (to reduce correlation) is possible. The incremental training of the network provides an opportunity to dynamically set the target outputs to give similar songs similar target outputs.

2.1 Explicitly Maintaining Entropy in Training

We train a neural network to take as input an audio spectrogram and to output a bin location where similar audio spectrograms will be hashed. We represent the outputs of the neural network in binary notation; for these experiments, we create a hash table with 1024 bins and therefore train the network with 10 binary target outputs.[1]

For training, we select 10 consecutive snippets from 1024 songs, sampled 116 ms apart (total 10,240 snippets). Each snippet from the same song is labeled with the same target song code. This is a weak label. Although the snippets that are temporally close may be 'similar', there is no guarantee that snippets that are further apart will be similar – even if they are from the same song. Moreover, snippets from different songs may be more similar than snippets from the same song. However, we do not require this detailed labeling; such labels would be infeasible to obtain for large sets.

The primary difficulty in training arises in finding suitable target outputs for each network. Every snippet of a song is labeled with the same target output. The target output for each song is assigned randomly, chosen without replacement from $P(S)$, where S is 10 bits – *i.e.,* each song is assigned a different set of 10 bits and $P(S)$ is the power set, containing 2^{10} different sets of 10 bits. Sampling without replacement is a crucial component of this procedure. Measured over the training set, the target outputs will have maximal entropy. Because we will be using the outputs as a hash, maintaining high entropy is crucial to preserving a good distribution in the hash bins. The drawback of this randomized target assignment is that different songs that sound similar may have entirely different output representations (large Hamming distance between their target outputs). If we force the network to learn these artificial distinctions, we may hinder or entirely prevent the network from being able to correctly perform the mapping.

[1] 1024 is a small number of bins for a hash table that will hold millions of items; in Section 2.3, we will show how to efficiently scale the size.

Instead of statically assigning the outputs, the target outputs shift throughout training. After every few epochs of weight updates, we measure the network's response to each of the training samples. We then dynamically reassign the target outputs for each song to the member from $P(S)$ that is closest to the network's response, aggregated over that song's snippets. During this process, we maintain two constraints: all snippets from each song must have the same output and no two songs can have the same target (so that each member of $P(S)$ is assigned once). This is a suitable procedure since the specific outputs of the network are not of interest, only the high-entropy distribution of them and the fact that same-song snippets map to the same outputs.

By letting the network adapt its outputs in this manner, the outputs across training examples can be effectively reordered to avoid forcing artificial distinctions. Through this process, the outputs are effectively being reordered to perform a weak form of clustering of songs: similar songs are likely to have small Hamming distances in target outputs. More details of a similar dynamic reordering can be found in [Caruana *et al.*, 1996]. The training procedure is summarized in Figure 2. Figure 3 shows the progress of the training procedure through epochs and compares it to training without output reordering[2]. Note that without reordering, the smaller network's performance was barely above random: 5.4/10. The larger network without reordering performs marginally better (5.6/10). However, both networks trained with output reordering learn a mapping more consistently than networks without reordering (7.0/10 and 7.5/10).

2.2 Using the Outputs as Hash-Keys

In this section, we take a first look at how the network's outputs perform when used as hash keys. There are two metrics that are of primary interest in measuring performance: (1) the number of candidates found in each hashed bin (that is, how many candidates must be considered at each lookup, indicating how well the candidates are distributed, with respect to queries) and (2) the number of hashed bins that include a snippet from the same song (indicating accuracy). In this section, we use a simple recall criterion: of the snippets in the hashed bin, are any from the correct song. Note that this is a very loose indicator of performance; in Section 3, when we utilize multiple hashes, we will make this constraint much tighter.

SELECT M=2^n songs and DEFINE $\boldsymbol{B}(t)$ = t^{th} binary code
FOREACH (song S_m)
| FOR (X_m iterations) { ADD snippet $S_{m,x}$ to training set }
| SELECT target $\boldsymbol{T}_m \in \{0,1\}^n$ s.t. $\boldsymbol{T}_{m1} = \boldsymbol{T}_{m2} \Leftrightarrow m_1 = m_2$
FOR (I_{dyn} iterations)
| TRAIN network for E_{dyn} epochs (E_{dyn} passes through data)
| FOREACH (song S_m)
| | SET $\boldsymbol{A}_m = \Sigma_{x \in X_m} \mathbf{O}(S_{m,x}) / X_m$
| | (where $\mathbf{O}(S_{m,x})$ = actual network output for snippet $S_{m,x}$)
| SET $U_t = \{ t \mid 0 < t \le M \}$ (the unassigned binary codes)
| SET $U_s = \{ s \mid 0 < s \le M \}$ (the unassigned songs)
| FOR (M iterations)
| | FIND $(s, t) = \arg\min_{s \in U_s, t \in U_t} \| \boldsymbol{B}(t) - \boldsymbol{A}_s \|_2$
| | SET $\boldsymbol{T}_s = \boldsymbol{B}(t)$
| | REMOVE s from U_s and REMOVE t from U_t
TRAIN network for E_{fixed} epochs

Settings used:
n=10 M= 1024, X_m =10 ($\forall m$), E_{dyn} =10, I_{dyn} =50, E_{fixed}=500

Figure 2: Training algorithm and parameter settings used.

[2] Standard back-propagation (BP) was used to train the networks [Hertz *et al.*, 1989]. Parameters for BP: learning rate 0.00025, momentum = 0.

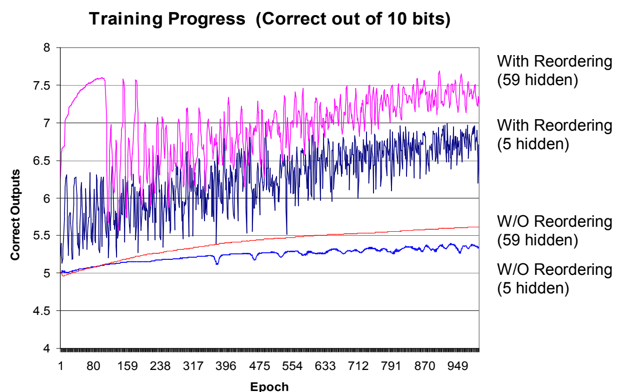

Figure 3: Training with and without output reordering, for small and large networks. Note that without output reordering, the number of correct outputs ~5.6/10. With reordering, it approaches 7.5.

For these exploratory experiments, we trained 58 networks, 2 with 5 hidden units, 2 with 7 hidden units, etc, up to 2 with 61 hidden units. Each was then used to hash approximately 38,000 snippets (drawn from 3,800 songs with 10 snippets each). These 3800 songs formed our test set for this stage, and were independent from the 1024-song training set. For this test, we removed the query q from the database and determined in which bin it would hash by propagating it through the networks. After propagation, for each of the 10 network outputs, if the output was greater than the median response of that output (as ascertained from the training set), it was assigned +1; otherwise, it was assigned 0. The median was chosen as the threshold to ensure that the network has an even distribution of 0/+1 responses. The 10 outputs were treated as a binary number representing the hash bin.

Figure 4 shows the total number of candidates (both correct and incorrect) found in the hashed bin (on average) as a function of the number of hidden units in the network. Figure 4 also shows the percentage of lookups that resulted in a bin which contained a snippet from the correct song. To understand Figure 4, the raw numbers are important to examine. Looking at the middle of the graph, every query on average was hashed into a bin with approximately 200 other snippets (~0.5% of the database). In ~78% of the retrievals, the hash bin contained a snippet from the correct song. Note that if the hashes were completely random, we would expect to find a snippet from the same song less than 1% of the time. Figure 4 also shows the negative effects of poorer training on the small networks. Returning to Figure 3, we see that even with the output reassignment, the small networks ("5 hidden") lags behind the large networks ("59

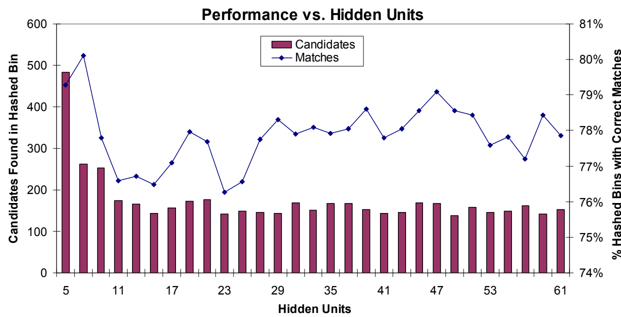

Figure 4: Performance of networks in terms of candidates found in hashed bin and % of queries finding matches from the same song in the bin to which it is hashed.

hidden") by about ½ bit of accuracy. The effect of the poorer training is seen in the larger number of candidates in the hashed bin coupled with the small absolute change in recall (from ~80% to ~76%) given by the networks with 5-9 hidden units (Figure 4). However, by using only networks with 11 or more hidden units, we can reduce the number of candidates by ~70% (from ~500 to ~150) with minor impact on the recall rate (< 2%).

In Figure 5, we compare the performance of using the outputs of the networks trained with and without output reassignment[3], and using networks with random weights. Note that using random weights is not the same as random hashing; by passing snippets through the network, it will be more likely to map similar snippets to the same bin; however, the weightings and combinations of the inputs will not be tuned to the task. As can be seen in Figure 5, for every lookup, there is a higher probability of finding a snippet from the same song using the fully trained networks, over networks with static-output training or with no training.

2.3 Using the Networks in Practice

In the previous section, we analyzed the performance of each of the networks in terms of the average number of candidates in the hashed bins and the accuracy of lookups. In this section, we combine the outputs of the networks to create a large-scale hashing system. So far, each of the networks was trained with only 10 outputs, allowing 1024 bins in the hash. A naïve way to scale the hash to a larger number of bins is to train the network with more outputs. However, because larger networks necessitate more training data, this becomes computationally prohibitive.[4]

Instead, we use two facts: that training networks is a randomized process dependent on the initial weights, and that networks with different architectures may learn different mappings. Hence, given that we have already trained multiple networks, we can select bits for our hash *from any of the networks' outputs*. Numerous methods can be used to select which outputs are used for a hash. We explored three methods of selecting the outputs: (1) random selection, (2) minimum-correlation selection, and (3) minimum-mutual-information selection. Both of the last two methods are chosen in a greedy manner [Chow & Liu, 1969]. By accounting for mutual information, we explicitly attempt to spread the samples across bins by ensuring that the combined entropy of the hash-index bits remains high. This method had a slight advantage over the others, so this will be used going forward.

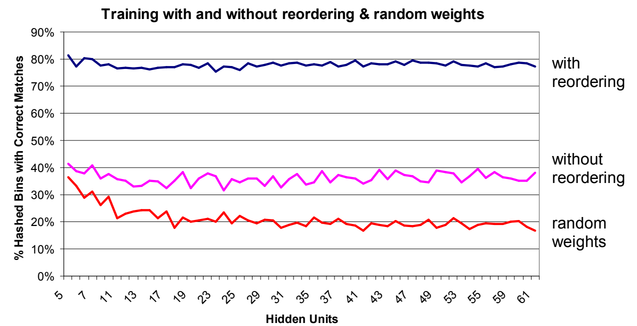

Figure 5: Performance of training procedures. Measured by % of queries finding the same song in the bin to which it is hashed. Avg # of snippets examined per bin for the training w/reordering is: 174; for static-output training: 123; and for random networks: 123.

We are no longer constrained to using only 10 bits since we can pick an arbitrary number of outputs from the network ensemble for our hash function. Figure 6 shows the performance in terms of the number of candidates in each of the hashed bins and correct matches, as a function of bits that are used for the hash (or equivalently, the number of total hash bins). Figure 6 also shows two other crucial results. The first is that by selecting the hash bits from multiple networks, we decrease the number of candidates even when using the same number of bins (1024). The number of candidates in each hashed bin decreased by 50% - from 200 (Figure 3) to 90 (Figure 4, column 1). Second, when increasing the number of bins from 1024 (2^{10}) to 4,194,304 (2^{22}), we see a decrease in the number of candidates from 90 to 5 per hashed bin. Although there is also a decrease in the number of correct matches, it is not proportional; it decreases by ~50% (instead of decreasing by 94% as does the number of candidates). In the next section, we describe how to take advantage of the small number of candidates and regain the loss in matches.

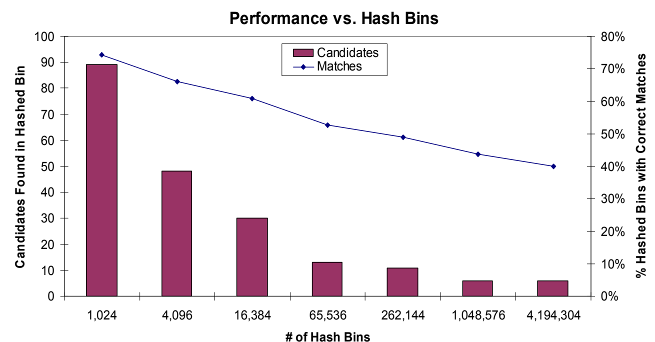

Figure 6: As we increase the number of bins by 4096×, the matches decrease by 50%. Meanwhile, the candidates decrease from ~90 to ~5.

[3] Fig. 3 provided a comparison of two training methods for two sizes of networks. Fig. 5 shows the effects of using the outputs of networks, trained with 3 methods, as hashes for the test data, as network size grows.

[4] Depending on # of hidden units, training ranged from hours to days.

3 A Complete Hashing System

From the previous sections, when a new query, *q*, arrives, it is passed into an ensemble of networks from which select outputs are used to determine a single hash location in a single hash table. Analogous to LSH, we can also generalize to *l* hash tables, with *l* distinct hash functions. To do this, we simply select from the unused outputs of the network ensemble to build another hash function that indexes into another hash table. Now, when *q* arrives, it is passed through all of the networks. The outputs of the networks, if they have been selected into one of *l* hashes, are used to determine the bin of that hash. In Figure 7, we experiment with the settings for *l* (1-22) and # of network outputs per hash (10-22: 1024 – 4,194,304 bins per hash).

In the previous section, we measured the recall as being whether the hashed bin contained a snippet from the correct song. Here, we tighten the definition of success to be the one we use in the final system. We rank order the snippets in the database according to how many of the *l* hash bins provided by the *l* hashes contain each snippet. The top ranked snippet is the one that occurs most frequently. We declare success if the top snippet comes from the same song as query *q*. *Note we never perform comparisons in the original high-dimensional spectrogram representation. After passing through the networks, the snippet is represented by the quantized binary outputs: the original representation is no longer needed.*

As can be seen in Figure 7A, the top line is the performance of the system with *l*=22 hashes; with only 1,024 bins, the number of candidates is unacceptably large: over 2,000. With smaller numbers of hashes (shown in the lower lines), the number of candidates decreases. As expected, looking towards the right of the graph, as the number of bins increases, the number of candidates decreases rapidly for all of the settings of *l* considered.

Figure 7B provides the most important results. For the top line (22 hashes), as the number of bins increases, the ability to find the best match barely decreases – despite the large drop in the number of candidates (Figure 7A). By examining only 200-250 candidates per query (~0.6% of the database), we achieve between 80-90% accuracy (*l* > 14).

In Figure 7B, note that for the lowest two lines (*l*=1,3), as the number of bins increases, the accuracy rises – in almost every other case, it decreases. The increase with *l*=1-3 occurs because there are a large number of ties (many songs have the same support) and we break ties with random se-

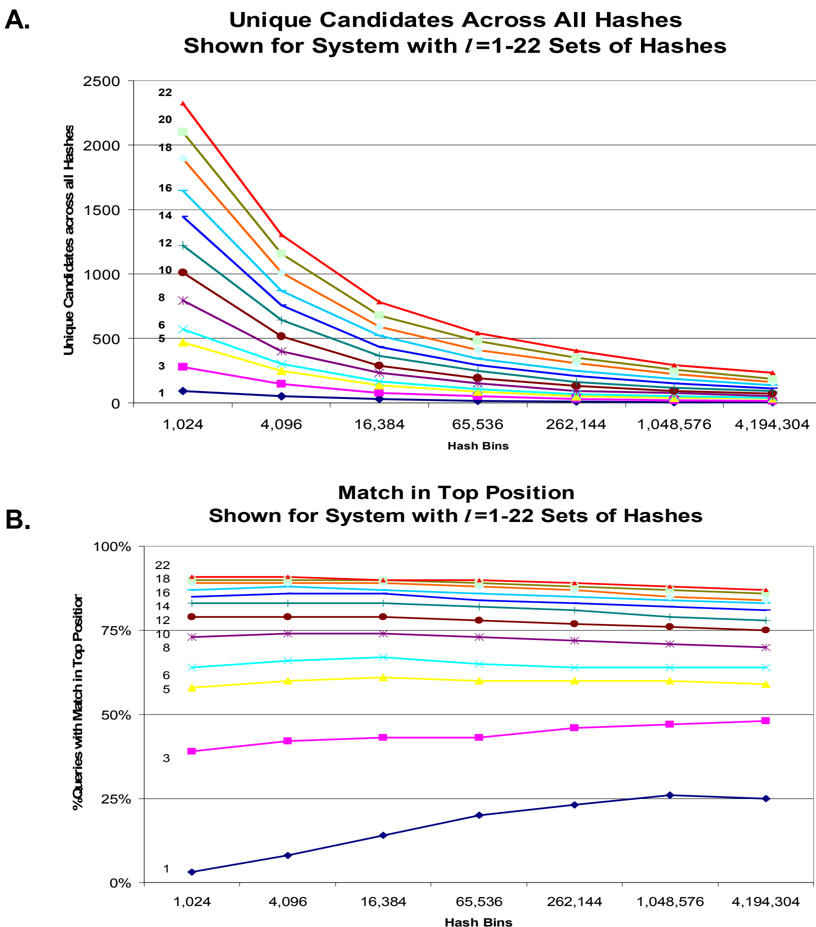

Figure 7: Performance of systems with (*l*=1..22) hashes, as a function of the number of bins.

A: # of candidates considered (smaller better).

B: % of times correct song identified (higher better).

lection. As the number of snippets hashed to the same bin decreases, the correct song competes with fewer incorrect ties and has a higher chance of top rank.

Finally, in Figure 7A, note that as the number of hashes increases, the number of candidates increases almost linearly. This likely indicates that the hashes are largely independent. If they were not independent, the number of unique candidates examined would overlap to a much greater degree. If there was significant repetition of candidates across the hashes, we would see the same "frequent-tie" phenomena that we commented on for *l*=1: the same songs would be repeatedly grouped and we would be unable to distinguish the correct co-occurrences from the accidental ones.

In summary, it is interesting to compare this procedure to LSH in terms of how each approach handles hashing approximate matches. LSH hashes only portions of the input vector to each of its multiple hashes. Intuitively, the goal is to ensure that if the points differ on some of the dimensions, by using only a few dimensions per hash, similar points will still be found in many of the hash bins. Our approach attempts to explicitly learn the similarities and use them to guide the hash function. Further, our approach allows similarity to be calculated on (potentially non-linear) transformations of the input rather than directly on the inputs. The use of multiple hashes in the system used here is to account for the imperfect learning of a difficult similarity functions.

4 Large Scale Experiments

In this section, we conduct a large-scale test of the system. The trained networks are used to hash 1,300,000 snippets. In this experiment, we use 6500 songs and 200 snippets from each song. As before, the snippets are of length 1.4 sec (128 11.6-ms slices). This spacing follows previous studies [Ke et *al.,* 2005; Haitsma & Kalker, 2002]. Our snippets are drawn approximately 116 ms apart.

As with our previous experiments, for every query snippet, we examine how many other snippets are considered as potential matches (the union of the *l* hashes), and also examine the number of times the top-ranked snippet came from the correct song. For these experiments, we attempted 46 different parameter settings, varying *l* and the number of bins per hash. Table I shows the performance for 5 groups of desired numbers of candidates; the first row (< 0.15%) indicates that if we want the average query to examine less than 0.15% of the database, what accuracy we can achieve, and with which parameter settings. In summary, we can find 1.4 sec snippets in a database of 6500 songs with 72% accuracy by using 2^{22} bins and *l*=18 sets of hashes – while only examining 0.27% of the database per query.

Table I: Best Results for 0.15 – 0.55% Candidates

% Candidates Examined	Best 3 Results (Accuracy, 2^{Bins}, *l* (sets))		
	top	second	third
< 0.15%	56% (22,10)	37% (20,5)	36% (22,5)
0.15 – 0.25%	69% (22,16)	66% (22,14)	62% (22,12)
0.25 – 0.35%	**72% (22,18)***	67% (20,14)	63% (20,12)
0.35 – 0.45%	73% (20,18)	70% (20,16)	63% (16,5)
0.45 – 0.55%	77% (20,22)	75% (20,20)	71%(18,16)

Beyond looking at the top match entry, Figure 8 demonstrates that by examining the top-25 snippets that appeared in the most bins, there is a smooth drop-off in finding the correct song beyond simply examining the single most frequently occurring snippet. Because the database contains 200 snippets from each song, multiple snippets can be found for each query.

Finally, we bring all of our results together and examine the performance of this system in the context of a longer song recognition system. We compare this system with a state-of-the art LSH-based system [Baluja & Covell, 2006], which extends [Ke et *al.,* 2005] by allowing fingerprints to be coarsely sampled to lower the memory required to recognize a large database of songs, while maintaining high recognition rates. Ke's system was an improvement over the *de facto* standard of [Haitsma & Kalker, 2002].

For these experiments, we used ***s*** sec of a song ($1.4 \leq \mathbf{s} \leq 25$) and integrated the evidence from individual snippet lookups to determine the correct song. Note that these experiments are *harder* than the hardest of the cases expected in practice. Figure 9 illustrates why: even though the [Baluja & Covell, 2006] system does not sample the database snippets as closely, their sampling of the probe snippets insures a database-to-nearest-probe misalignment of 23.2 ms, at most, at each database snippet: at least one of the

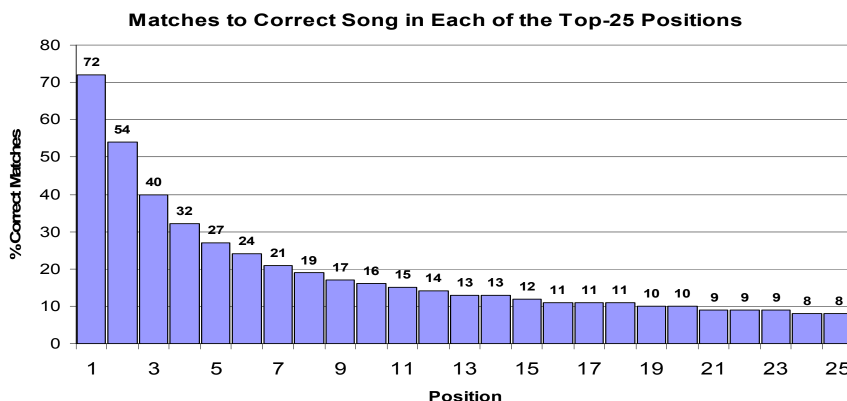

Figure 8: Song prediction for each of the top-25 matches. Here, position 1 = 72%, position 2=54%, position 25 = 8%.

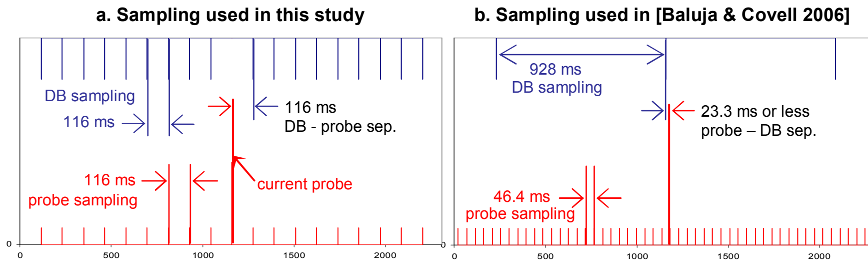

Figure 9: Sampling used to generate Table II. The sampling used was chosen to test worst-case interactions between probe and database snippets. For that table, each probe snippet is explicitly deleted from the database before snippet retrieval. The result is the probe snippets are ***all*** 116 ms away from the nearest database snippet. This test is more difficult than the sampling offsets for [Baluja & Covell 2006], since that system always encounters many snippets with small probe-database separations.

probes will be at least that close, since the probe density is 46.4 ms. By removing the matching snippet from the database, our experiments force ***all*** database-to-probe misalignments to be a full 116 ms, 5 times more than the closest sampling of [Baluja & Covell, 2006]. We do not rely on dense probe sampling to guarantee nearby (in-time) matches. Results are shown in Table II.

Table II: Performance vs. Query Length (in seconds) (with sampling shown in Figure 9a)

	1.4s	2s	5s	13s	25s
Forgiving-Hash	72.3%	89.4%	98.9%	99.2%	99.5%
LSH-System	35.5%	51.6%	73.0%	98.4%	98.6%

For the forgiving hasher, we use 2^{22} bins & l=18 hashes. For the LSH system, we set the parameters individually for each trial to maximize performance while allowing approximately equal hash-table lookups. Unlike [Baluja & Covell 2006], we count the candidates returned from each hash-table, not just the candidates that passed the hash-voting thresholds and thereby triggered retrieval of the original fingerprint. We use this metric on the LSH system since under forgiving hashing, there is no retrieval of an original fingerprint. Each candidate simply has a counter associated with it; neither the spectrogram, nor a representation of the spectrogram, is ever compared with each candidate. In summary, for our system, near perfect accuracy at even 5 sec. is achieved while considering only a tiny fraction of the candidates.

5 Conclusions & Future Work

We have presented a system that surpasses the state of the art, both in terms of efficiency and accuracy, for retrieval in high-dimensional spaces where similarity is not well-defined. The forgiving hasher ignores the small differences in snippets and maps them to the same bin by learning a similarity function from only weakly labeled positive examples. The system is designed to work with video and images as well; those experiments are currently underway.

Beyond further experiments with scale, there are many directions for future work. The number of bits trained per learner should be explored; by training only 1 bit, numerous binary classifiers become usable. To ensure high entropy for unseen songs, training with a song set that "spans" the space is desirable; this may be approximated by training learners with different songs. Currently, the system handles approximate-match retrieval which is similar to handling noise. Tests are currently being conducted to quantify the resilience to structured and unstructured noise. Finally, it will be interesting to test forgiving hashing on music-genre identification, where 'similarity' is even less well-defined.

References

[Baluja & Covell, 2006] S. Baluja, M. Covell. Content Fingerprinting with Wavelets. *Third Conference on Visual Media Production (CVMP).*

[Bar Hillel *et al.*, 2003] A. Bar-Hillel, T. Hertz, N. Shental, D. Weinshall. Learning Distance Functions using Equivalence Relations. *International Conference on Machine Learning.*

[Burges *et al.*, 2003] J. C. Burges, J. C. Platt, S. Jana. Distortion Discriminant Analysis for Audio Fingerprinting. *IEEE Trans. Speech and Audio Processing*, vol. 11.

[Caruana *et al*, 1996] R. Caruana, S. Baluja, T. Mitchell. Using the future to "sort out" the present: Rankprop and MTL. *Neural Information Processing Systems 8.*

[Chow & Liu, 1968] C. Chow, C. Liu. Approximating discrete probability distributions via dependence trees. *IEEE Trans. Info. Theory,* vol. IT-14.

[Gionis *et al.*, 1999] A Gionis, P. Indyk, R. Motwani. Similarity search in high dimensions via hashing. *25th Very Large Data Base Conference.*

[Haitsma and Kalker, 2002] J. Haitsma, T. Kalker. A Highly Robust Audio Fingerprinting System. *International Conference on Music Information Retrieval*.

[Hastie & Tibshirani, 1996] T. Hastie, R. Tibshirani. Discriminant Adaptive Nearest Neighbor, *IEEE PAMI,* vol. 18.

[Hertz *et al.*, 1991] J. Hertz, A. Krogh, R. Palmer. *Introduction to the Theory of Neural Computing* (Addison-Welsey).

[Ke *et al.,* 2005] Y. Ke, D. Hoiem, R. Sukthankar. Computer Vision for Music Identification, *Computer Vision and Pattern Recognition*.

[Pollack, 1990] J.B. Pollack. Recursive Distributed Representations, *Artificial Intelligence*, vol. 46.

[Shakhnarovich, 2006] G. Shakhnarovich. *Learning Task Specific Similarity.* Ph.D. Thesis, MIT.

[Shazam, 2005] Shazam Entertainment http://www.shazamentertainment.com

[Shental *et al.*, 2002] N. Shental, T. Hertz, D. Weinshall, M. Pavel. Adjustment Learning and Relevant Component Analysis. *ECCV.*

[Tsang *et al.*, 2005] I. W. Tsang, P.-M. Cheung, J. T. Kwok. Kernel Relevant Component Analysis for Distance Metric Learning. *IJCNN.*