Advanced Topics in Computer Vision and Image Processing

Lecture 02
The Image Classification Problem
Military College of Signals

Asim D. Bakhshi
asim.dilawar@mcs.edu.pk

October 30, 2021

Contents

1 Motivation for Image Classification **2**
1.1 The Image Model 2
1.2 Challenges 2

2 The Data-Driven Approach **3**
2.1 Heuristics 3
2.2 The Image Classification Pipeline 4
2.3 Should you collect your own dataset? 4

3 Nearest Neighbour Classification **5**
3.1 Description of CIFAR10 Dataset 5
3.2 Classification Approach 5

3.3 Distance Metric 6
3.4 API and the Evaluation Criterion of the Classifier 7
3.5 L1 vs. L2 7
3.6 k-Nearest Neighbour Classification 8

4 Validation Set and Hyperparameter Tuning 8
4.1 Cross-validation Procedure 9

5 Strengths and Limitations of Nearest Neighbour Classification 11
5.1 Heuristically and Practically Simple 11
5.2 Cheap to Train; Expensive to Test 11
5.3 Curse of Dimensionality 11

6 Summary 13

1 Motivation for Image Classification

Image classification is one of the core problems in Computer Vision that, despite its simplicity, has a large variety of practical applications. Many other seemingly distinct Computer Vision tasks (such as object detection, segmentation) can be reduced to image classification.

1.1 The Image Model

The image classification model in Figure 1 takes a single image tensor $\mathbf{I} \in \mathbb{R}^{m \times n \times c}$ and assigns probabilities $p \in [0, 1]$ to 4 labels, cat, dog, hat, mug. As shown in the figure, image is represented as one large 3-dimensional array of numbers. In this example, the cat image is $m = 248$ pixels wide, $n = 400$ pixels long, and has $c = 3$ color channels Red, Green, Blue (or RGB for short). Therefore, the image consists of 248 x 400 x 3 numbers, or a total of 297,600 numbers. Each number belong to an integer space $\mathbb{R}$ that ranges from 0 (black) to 255 (white). Our task is to turn this quarter of a million numbers into a single label, such as “cat”.

1.2 Challenges

Since this task of recognizing a visual concept (e.g. cat) is relatively trivial for a human to perform, it is worth considering the challenges involved from the perspective of a Computer Vision algorithm. The list given below is by no means exhaustive.

1. ***Viewpoint variation.*** A single instance of an object can be oriented in many ways with respect to the camera.

2. ***Scale variation.*** Visual classes often exhibit variation in their size (size in the real world, not only in terms of their extent in the image).

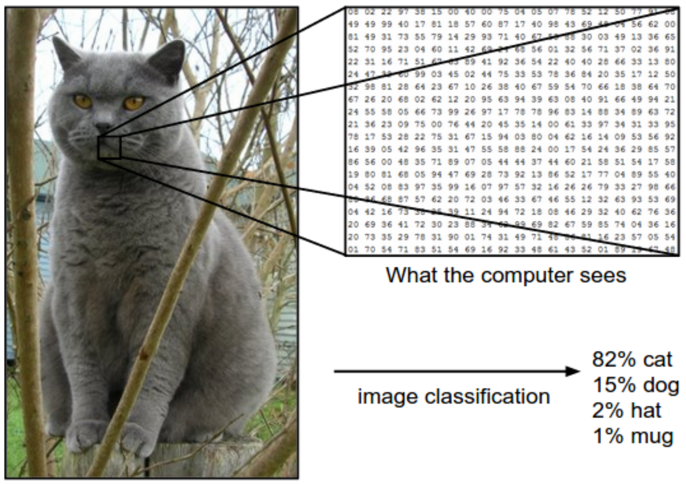

Figure 1: Simplified Image Classification Model

3. ***Deformation.*** Many objects of interest are not rigid bodies and can be deformed in extreme ways.
4. ***Occlusion.*** The objects of interest can be occluded. Sometimes only a small portion of an object (as little as few pixels) could be visible.
5. ***Illumination conditions.*** The effects of illumination are drastic on the pixel level.
6. ***Background clutter.*** The objects of interest may blend into their environment, making them hard to identify.
7. ***Intra-class variation.*** The classes of interest can often be relatively broad, such as chair. There are many different types of these objects, each with their own appearance.

A good image classification model must be invariant to the cross product of all these variations, while simultaneously retaining sensitivity to the inter-class variations.

2 The Data-Driven Approach

2.1 Heuristics

How might we go about writing an algorithm that can classify images into distinct categories? Unlike writing an algorithm for, for example, sorting a list of numbers, it is not obvious how one might write an algorithm for identifying cats in images [1]. Therefore, instead of trying to specify what every one of the

[1]Check out this one if you are curious [1]

categories of interest look like directly in code, the approach that we will take is not unlike one you would take with a child: we're going to provide the computer with many examples of each class and then develop learning algorithms that look at these examples and learn about the visual appearance of each class. This approach is referred to as a data-driven approach, since it relies on first accumulating a training dataset of labeled images.

2.2 The Image Classification Pipeline

We have seen that the task in Image Classification is to take an array of pixels that represents a single image and assign a label to it. Our complete pipeline can be formalized as follows:

1. **Input:** Our input consists of a set of N images, each labeled with one of K different classes. We refer to this data as the training set.

2. **Learning:** Our task is to use the training set (a subset of N) to learn what every one of the classes looks like. We refer to this step as training a classifier, or learning a model.

3. **Evaluation:** In the end, we evaluate the quality of the classifier by asking it to predict labels for a new set of images that it has never seen before. We will then compare the true labels of these images to the ones predicted by the classifier. Intuitively, we're hoping that a lot of the predictions match up with the true answers (which we call the ground truth).

2.3 Should you collect your own dataset?

Fortunately, there are so many datasets that you might not need to do so unless you have a specific requirement that is not fulfilled by these. Here is a list of some of the datasets which are handy for most of the computer vision problems.

- Labelme: an online annotation tool to build image databases for computer vision research.

- OpenSurfaces: a large database of annotated surfaces created from real-world consumer photographs.

- ImageNet: a large-scale image dataset for visual recognition organized by WordNet hierarchy.

- SUN Database: a benchmark for scene recognition and object detection with annotated scene categories and segmented objects.

- Places Database: a scene-centric database with 205 scene categories and 2.5 millions of labelled images.

- NYU Depth Dataset v2: a RGB-D dataset of segmented indoor scenes.

- Microsoft COCO: a new benchmark for image recognition, segmentation and captioning.
- Flickr100M: 100 million creative commons Flickr images
- Labeled Faces in the Wild: a dataset of 13,000 labeled face photographs.
- Human Pose Dataset: a benchmark for articulated human pose estimation.
- YouTube Faces DB: a face video dataset for unconstrained face recognition in videos.
- UCF101: an action recognition data set of realistic action videos with 101 action categories.
- HMDB-51: a large human motion dataset of 51 action classes.
- CIFAR10/100: labelled datasets of 60000 images each from 10 and 100 categories respectively.

3 Nearest Neighbour Classification

As our first approach, we will develop what we call a Nearest Neighbor Classifier. This classifier has nothing to do with Convolutional Neural Networks and it is very rarely used in practice, but it will allow us to get an idea about the basic approach to an image classification problem.

3.1 Description of CIFAR10 Dataset

One popular toy image classification dataset is the CIFAR-10 dataset [2]. This dataset consists of 60,000 tiny images that are 32 pixels high and wide ($m = n = 32$). Each image is labeled with one of $K = 10$ classes (for example "airplane, automobile, bird, etc"). These 60,000 images are partitioned into a training set of $N_{training} = 50000$ images and a test set of $N_{test} = 10,000$ images. In the Figure 2 below you can see 10 random example images from each one of the 10 classes.

3.2 Classification Approach

Suppose now that we are given the CIFAR-10 training set of 50,000 images (5,000 images for every one of the labels), and we wish to label the remaining 10,000. The nearest neighbor classifier will take a test image, compare it to every single one of the training images, and predict the label of the closest training image.

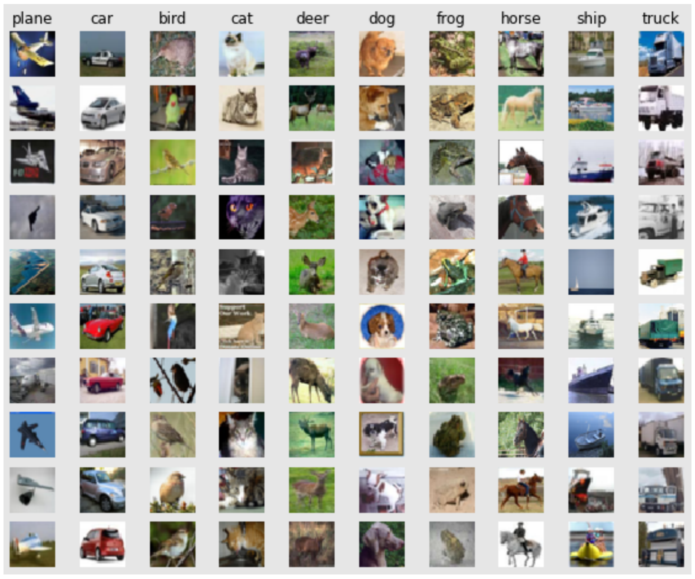

Figure 2: 10 Random Samples from the Data

3.3 Distance Metric

We must now ask ourselves how exactly we should compare two images, which in this case are just two tensors of dimensions $32 \times 32 \times 3$. One of the simplest possibilities is to compare the images pixel by pixel and add up all the differences. In other words, given two images and representing them as flattened column vectors I_1, I_2 a reasonable choice for comparing them might be the **L1 distance** given by

$$d_1(I_1, I_2) = \sum_n |I_1^n - I_2^n| \tag{1}$$

where the sum is taken over all pixels n. The procedure is visualized below in Figure 3:

There are many other ways of computing distances between vectors. Another common choice could be to instead use the **L2 distance**, which has the geometric interpretation of computing the euclidean distance between two vectors. The distance takes the form:

$$d_2(I_1, I_2) = \sqrt{\sum_n (I_1^n - I_2^n)^2} \tag{2}$$

In other words we would be computing the pixelwise difference as before, but this time we square all of them, add them up and finally take the square root.

$$\left| \overset{\text{test image}}{\begin{array}{|c|c|c|c|}\hline 56 & 32 & 10 & 18 \\ \hline 90 & 23 & 128 & 133 \\ \hline 24 & 26 & 178 & 200 \\ \hline 2 & 0 & 255 & 220 \\ \hline \end{array}} - \overset{\text{training image}}{\begin{array}{|c|c|c|c|}\hline 10 & 20 & 24 & 17 \\ \hline 8 & 10 & 89 & 100 \\ \hline 12 & 16 & 178 & 170 \\ \hline 4 & 32 & 233 & 112 \\ \hline \end{array}} \right| = \overset{\text{pixel-wise absolute value differences}}{\begin{array}{|c|c|c|c|}\hline 46 & 12 & 14 & 1 \\ \hline 82 & 13 & 39 & 33 \\ \hline 12 & 10 & 0 & 30 \\ \hline 2 & 32 & 22 & 108 \\ \hline \end{array}} \rightarrow 456$$

Figure 3: An example of using pixel-wise differences to compare two images with L1 distance (for one color channel in this example). Two images are subtracted elementwise and then all differences are added up to a single number. If two images are identical the result will be zero. But if the images are very different the result will be large.

3.4 API and the Evaluation Criterion of the Classifier

As an evaluation criterion, it is common to use the accuracy, which measures the fraction of predictions that were correct. Notice that all classifiers we will build satisfy this one common API: they have a **train(X,y)** function that takes the data and the labels to learn from. Internally, the class should build some kind of *model* of the labels and how they can be predicted from the data. And then there is a **predict(X)** function, which takes new data and predicts the labels.

If you implement the nearest neighbour classifier, you would see that this classifier only achieves 38.6% on CIFAR-10. That's more impressive than guessing at random (which would give 10% accuracy since there are 10 classes), but nowhere near human performance (which is estimated at about 94%) or near state-of-the-art Convolutional Neural Networks that achieve about 95%, matching human accuracy.

In a practical nearest neighbor application we could leave out the square root operation because square root is a monotonic function. That is, it scales the absolute sizes of the distances but it preserves the ordering, so the nearest neighbors with or without it are identical. If you ran the Nearest Neighbor classifier on CIFAR-10 with this distance, you would obtain 35.4% accuracy (slightly lower than our L1 distance result).

3.5 L1 vs. L2

It is interesting to consider differences between the two metrics. In particular, the L2 distance is much more unforgiving than the L1 distance when it comes to differences between two vectors. That is, the L2 distance prefers many medium disagreements to one big one. L1 and L2 distances (or equivalently the L1/L2 norms of the differences between a pair of images) are the most commonly used special cases of a p-norm.

3.6 k-Nearest Neighbour Classification

You may have noticed that it is strange to only use the label of the nearest image when we wish to make a prediction. Indeed, it is almost always the case that one can do better by using what's called a k-Nearest Neighbor Classifier. The idea is very simple: instead of finding the single closest image in the training set, we will find the top k closest images, and have them vote on the label of the test image. In particular, when $k = 1$, we recover the Nearest Neighbor classifier. Intuitively, higher values of k have a smoothing effect that makes the classifier more resistant to outliers as shown in Figure 4

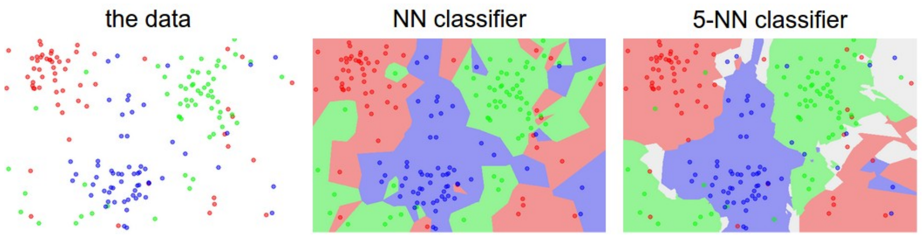

Figure 4: An example of the difference between Nearest Neighbor and a 5-Nearest Neighbor classifier, using 2-dimensional points and 3 classes (red, blue, green). The colored regions show the decision boundaries induced by the classifier with an L2 distance. The white regions show points that are ambiguously classified (i.e. class votes are tied for at least two classes). Notice that in the case of a NN classifier, outlier datapoints (e.g. green point in the middle of a cloud of blue points) create small islands of likely incorrect predictions, while the 5-NN classifier smooths over these irregularities, likely leading to better generalization on the test data (not shown). Also note that the gray regions in the 5-NN image are caused by ties in the votes among the nearest neighbors (e.g. 2 neighbors are red, next two neighbors are blue, last neighbor is green).

In practice, you will almost always want to use k-Nearest Neighbor. But what value of k should you use? We turn to this problem next.

4 Validation Set and Hyperparameter Tuning

The k-nearest neighbor classifier requires a setting for k. But what number works best? Additionally, we saw that there are many different distance functions we could have used: L1 norm, L2 norm, there are many other choices we didn't even consider (e.g. dot products). These choices are called hyperparameters and they come up very often in the design of many Machine Learning algorithms that learn from data. It's often not obvious what values/settings one should choose.

You might be tempted to suggest that we should try out many different values and see what works best. That is a fine idea and that's indeed what we will do, but this must be done very carefully. In particular, **we cannot use the test set for the purpose of tweaking hyperparameters**. Whenever you're designing Machine Learning algorithms, you should think of the test set as a very precious resource that should ideally never be touched until one time at the very end. Otherwise, the very real danger is that you may tune your hyperparameters to work well on the test set, but if you were to deploy your model you could see a significantly reduced performance. In practice, we would say that you **overfit to the test set**. Another way of looking at it is that if you tune your hyperparameters on the test set, you are effectively using the test set as the training set, and therefore the performance you achieve on it will be too optimistic with respect to what you might actually observe when you deploy your model. But if you only use the test set once at end, it remains a good proxy for measuring the generalization of your classifier.

Luckily, there is a correct way of tuning the hyperparameters and it does not touch the test set at all. The idea is to split our training set in two: a slightly smaller training set, and what we call a validation set. Using CIFAR-10 as an example, we could for example use 49000 of the training images for training, and leave 1000 aside for validation. This validation set is essentially used as a fake test set to tune the hyper-parameters.

4.1 Cross-validation Procedure

In cases where the size of your training data (and therefore also the validation data) might be small, people sometimes use a more sophisticated technique for hyperparameter tuning called cross-validation. Working with our previous example, the idea is that instead of arbitrarily picking the first 1000 datapoints to be the validation set and rest training set, you can get a better and less noisy estimate of how well a certain value of k works by iterating over different validation sets and averaging the performance across these. For example, in 5-fold cross-validation, we would split the training data into 5 equal folds, use 4 of them for training, and 1 for validation. We would then iterate over which fold is the validation fold, evaluate the performance, and finally average the performance across the different folds.

In practice, people prefer to avoid cross-validation in favor of having a single validation split, since cross-validation can be computationally expensive. The splits people tend to use is between 50%-90% of the training data for training and rest for validation. However, this depends on multiple factors: For example if the number of hyperparameters is large you may prefer to use bigger validation splits. If the number of examples in the validation set is small (perhaps only a few hundred or so), it is safer to use cross-validation. Typical number of folds you can see in practice would be 3-fold, 5-fold or 10-fold cross-validation.

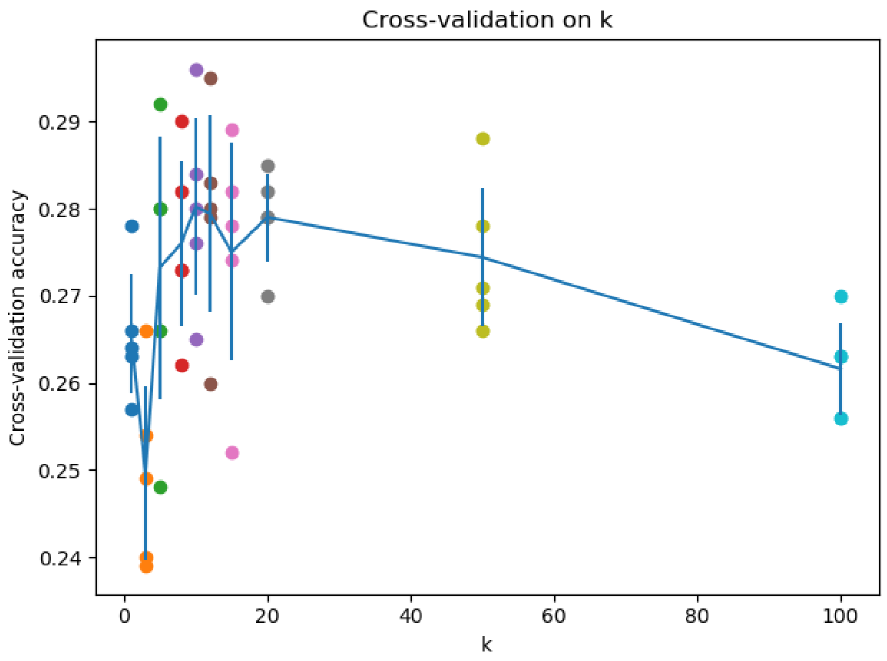

Figure 5: Example of a 5-fold cross-validation run for the parameter k. For each value of k we train on 4 folds and evaluate on the 5th. Hence, for each k we receive 5 accuracies on the validation fold (accuracy is the y-axis, each result is a point). The trend line is drawn through the average of the results for each k and the error bars indicate the standard deviation. Note that in this particular case, the cross-validation suggests that a value of about $k = 7$ works best on this particular dataset (corresponding to the peak in the plot). If we used more than 5 folds, we might expect to see a smoother (i.e. less noisy) curve.

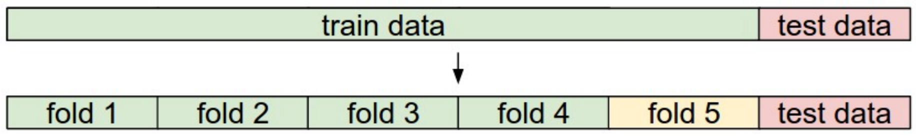

Figure 6: Common data splits. A training and test set is given. The training set is split into folds (for example 5 folds here). The folds 1-4 become the training set. One fold (e.g. fold 5 here in yellow) is denoted as the Validation fold and is used to tune the hyperparameters. Cross-validation goes a step further and iterates over the choice of which fold is the validation fold, separately from 1-5. This would be referred to as 5-fold cross-validation. In the very end once the model is trained and all the best hyperparameters were determined, the model is evaluated a single time on the test data (red).

5 Strengths and Limitations of Nearest Neighbour Classification

It is worth considering some advantages and drawbacks of the Nearest Neighbor classifier.

5.1 Heuristically and Practically Simple

Clearly, one advantage is that it is very simple to implement and understand. Additionally, the classifier takes no time to train, since all that is required is to store and possibly index the training data.

5.2 Cheap to Train; Expensive to Test

However, we pay that computational cost at test time, since classifying a test example requires a comparison to every single training example. This is backwards, since in practice we often care about the test time efficiency much more than the efficiency at training time. In fact, the deep neural networks we will develop later in this class shift this trade-off to the other extreme: They are very expensive to train, but once the training is finished it is very cheap to classify a new test example. This mode of operation is much more desirable in practice.

As an aside, the computational complexity of the Nearest Neighbor classifier is an active area of research, and several Approximate Nearest Neighbor (ANN) algorithms and libraries exist that can accelerate the nearest neighbor lookup in a dataset (e.g. FLANN) [3]. These algorithms allow one to trade off the correctness of the nearest neighbor retrieval with its space/time complexity during retrieval, and usually rely on a pre-processing/indexing stage that involves building a kdtree, or running the k-means algorithm.

5.3 Curse of Dimensionality

The Nearest Neighbor Classifier may sometimes be a good choice in some settings (especially if the data is low-dimensional), but it is rarely appropriate for use in practical image classification settings. One problem is that images are high-dimensional objects (i.e. they often contain many pixels), and distances over high-dimensional spaces can be very counter-intuitive. Figure 7 illustrates the point that the pixel-based L2 similarities we developed above are very different from perceptual similarities.

Figure 8 shows a visualization to convince you that using pixel differences to compare images is inadequate. We can use a visualization technique called t-SNE [4] to take the CIFAR-10 images and embed them in two dimensions so that their (local) pairwise distances are best preserved. In this visualization, images that are shown nearby are considered to be very near according to the L2 pixel-wise distance we developed above:

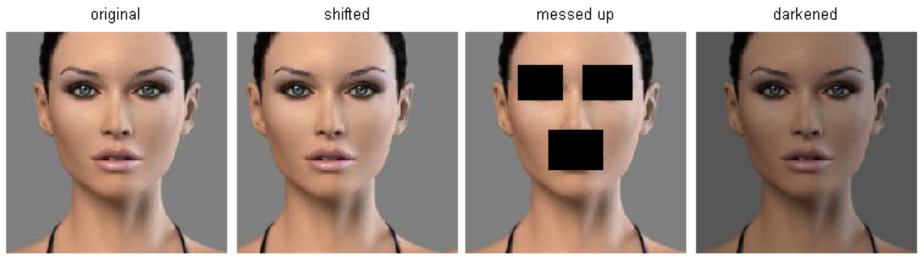

Figure 7: Pixel-based distances on high-dimensional data (and images especially) can be very unintuitive. An original image (left) and three other images next to it that are all equally far away from it based on L2 pixel distance. Clearly, the pixel-wise distance does not correspond at all to perceptual or semantic similarity.

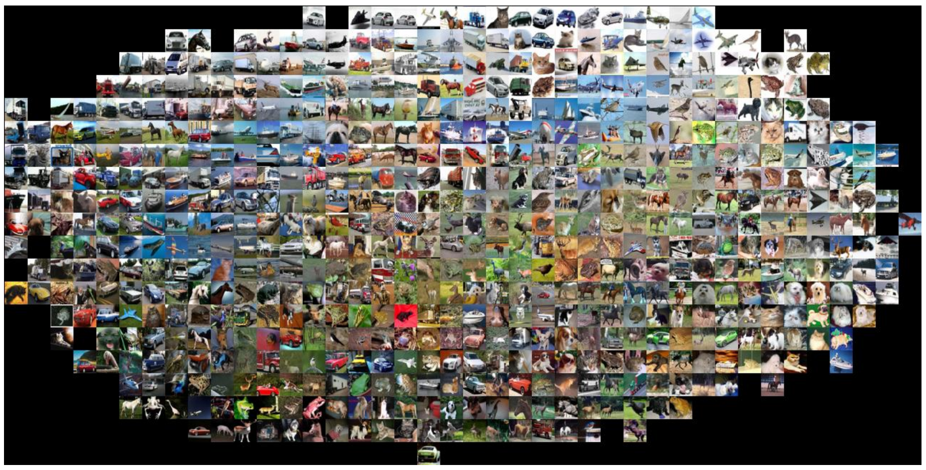

Figure 8: CIFAR-10 images embedded in two dimensions with t-SNE [4]. Images that are nearby on this image are considered to be close based on the L2 pixel distance. Notice the strong effect of background rather than semantic class differences. Click here for a bigger version of this visualization.

In particular, note that images that are nearby each other are much more a function of the general color distribution of the images, or the type of background rather than their semantic identity. For example, a dog can be seen very near a frog since both happen to be on white background. Ideally we would like images of all of the 10 classes to form their own clusters, so that images of the same class are nearby to each other regardless of irrelevant characteristics and variations (such as the background). However, to get this property we will have to go beyond raw pixels.

6 Summary

- We introduced the problem of Image Classification, in which we are given a set of images that are all labeled with a single category. We are then asked to predict these categories for a novel set of test images and measure the accuracy of the predictions.
- We introduced a simple classifier called the Nearest Neighbor classifier. We saw that there are multiple hyper-parameters (such as value of k, or the type of distance used to compare examples) that are associated with this classifier and that there was no obvious way of choosing them.
- We saw that the correct way to set these hyperparameters is to split your training data into two: a training set and a fake test set, which we call validation set. We try different hyperparameter values and keep the values that lead to the best performance on the validation set.
- If the lack of training data is a concern, we discussed a procedure called cross-validation, which can help reduce noise in estimating which hyperparameters work best.
- Once the best hyperparameters are found, we fix them and perform a single evaluation on the actual test set.
- We saw that Nearest Neighbor can get us about 40% accuracy on CIFAR-10. It is simple to implement but requires us to store the entire training set and it is expensive to evaluate on a test image.
- Finally, we saw that the use of L1 or L2 distances on raw pixel values is not adequate since the distances correlate more strongly with backgrounds and color distributions of images than with their semantic content.
- In next lectures we will embark on addressing these challenges and eventually arrive at solutions that give 90% accuracies, allow us to completely discard the training set once learning is complete, and they will allow us to evaluate a test image in less than a millisecond.

References

[1] John Canny. “A computational approach to edge detection”. *IEEE Transactions on pattern analysis and machine intelligence*, pp. 679–698, 1986.

[2] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. 2009.

[3] Marius Muja and David G Lowe. “Fast approximate nearest neighbors with automatic algorithm configuration.” *VISAPP (1)* 2, p. 2, 2009.

[4] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” *Journal of machine learning research* 9, 2008.