Lecture 2. First Approaches for Image Classification

Disclaimer: This note was modified from cs231n lecture notes by Prof. Li Fei-Fei at Stanford University.

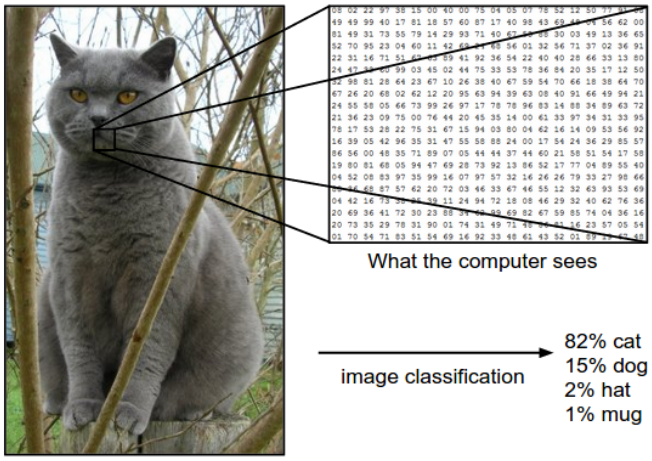
This is an introductory lecture designed to introduce people from outside of Computer Vision to the Image Classification problem, and the data-driven approach.

The Table of Contents:

Image Classification Task

Motivation. In this section we will introduce the Image Classification problem, which is the task of assigning an input image one label from a fixed set of categories. This is one of the core problems in Computer Vision that, despite its simplicity, has a large variety of practical applications. Moreover, as we will see later in the course, many other seemingly distinct Computer Vision tasks (such as object detection, segmentation) can be reduced to image classification.

Example. For example, in the image below an image classification model takes a single image and assigns probabilities to 4 labels, *{cat, dog, hat, mug}*. As shown in the image, keep in mind that to a computer an image is represented as one large 3-dimensional array of numbers. In this example, the cat image is 248 pixels wide, 400 pixels tall, and has three 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 is an integer 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"*.

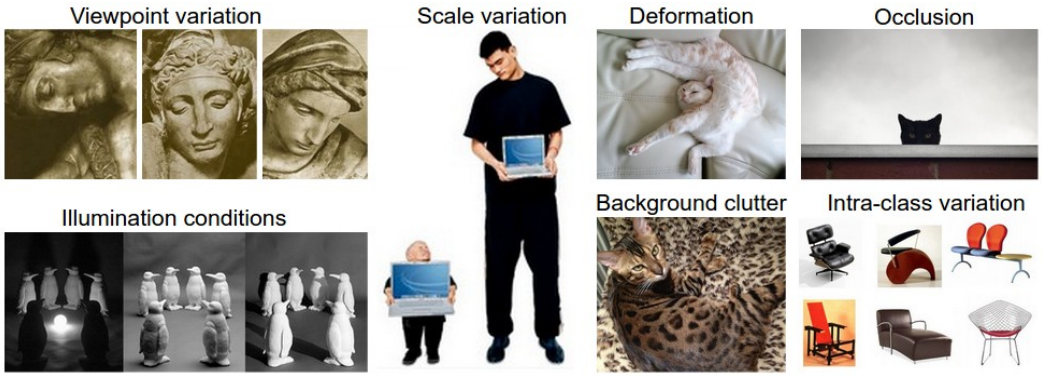


The task in Image Classification is to predict a single label (or a distribution over labels as shown here to indicate our confidence) for a given image. Images are 3-dimensional arrays of integers from 0 to 255, of size Width x Height x 3. The 3 represents the three color channels Red, Green, Blue.

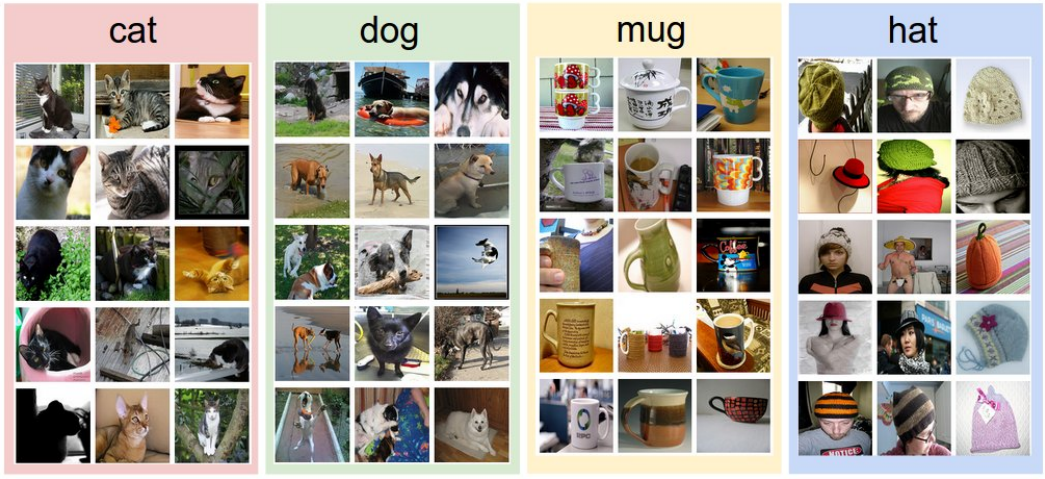
Challenges. Since this task of recognizing a visual concept (e.g. cat) is relatively trivial for a human to perform, it is worth considering the challenges involved from the perspective of a Computer Vision algorithm. As we present (an inexhaustive) list of challenges below, keep in mind the raw representation of images as a 3-D array of brightness values:

- **Viewpoint variation**. A single instance of an object can be oriented in many ways with respect to the camera.
- **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).
- **Deformation**. Many objects of interest are not rigid bodies and can be deformed in extreme ways.
- **Occlusion**. The objects of interest can be occluded. Sometimes only a small portion of an object (as little as few pixels) could be visible.
- **Illumination conditions**. The effects of illumination are drastic on the pixel level.
- **Background clutter**. The objects of interest may *blend* into their environment, making them hard to identify.
- **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.



Data-driven approach. 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. Therefore, instead of trying to specify what every one of the 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. Here is an example of what such a dataset might look like:



An example training set for four visual categories. In practice we may have thousands of categories and hundreds of thousands of images for each category.

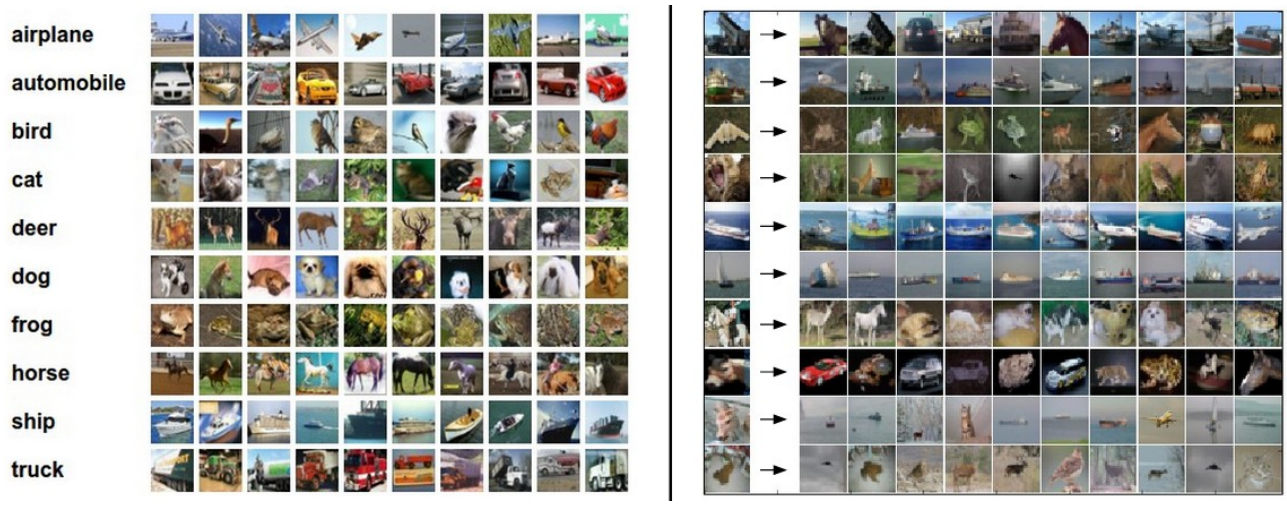
The image classification pipeline. We've seen that the task in Image Classification is to take an array of pixels that represents a single image and assign a label to it. Our complete pipeline can be formalized as follows:

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

Nearest Neighbor Classifiers

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.

Example image classification dataset: CIFAR-10. One popular toy image classification dataset is the CIFAR-10 dataset. This dataset consists of 60,000 tiny images that are 32 pixels high and wide. Each image is labeled with one of 10 classes (for example *"airplane, automobile, bird, etc"*). These 60,000 images are partitioned into a training set of 50,000 images and a test set of 10,000 images. In the image below you can see 10 random example images from each one of the 10 classes:



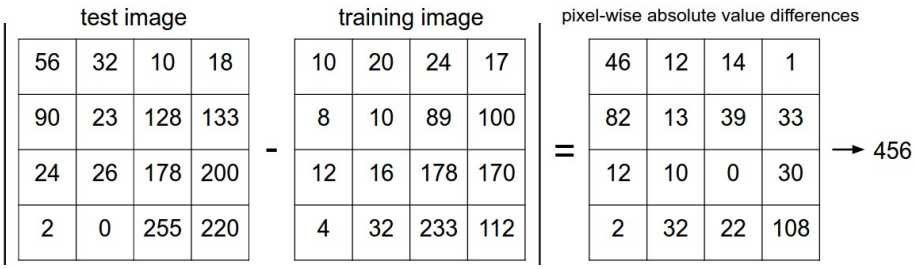
Left: Example images from the CIFAR-10 dataset. Right: first column shows a few test images and next to each we show the top 10 nearest neighbors in the training set according to pixel-wise difference.

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. In the image above and on the right you can see an example result of such a procedure for 10 example test images. Notice that in only about 3 out of 10 examples an image of the same class is retrieved, while in the other 7 examples this is not the case. For example, in the 8th row the nearest training image to the horse head is a red car, presumably due to the strong black background. As a result, this image of a horse would in this case be mislabeled as a car.

You may have noticed that we left unspecified the details of exactly how we compare two images, which in this case are just two blocks of 32 x 32 x 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 vectors I_1, I_2 , a reasonable choice for comparing them might be the **L1 distance**:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

Where the sum is taken over all pixels. Here is the procedure visualized:



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.

Let's also look at how we might implement the classifier in code. First, let's load the CIFAR-10 data into memory as 4 arrays: the training data/labels and the test data/labels. In the code below, `Xtr` (of size 50,000 x 32 x 32 x 3) holds all the images in the training set, and a corresponding 1-dimensional array `Ytr` (of length 50,000) holds the training labels (from 0 to 9):

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

Now that we have all images stretched out as rows, here is how we could train and evaluate a classifier:

```
nn = NearestNeighbor() # create a Nearest Neighbor classifier class
nn.train(Xtr_rows, Ytr) # train the classifier on the training images and labels
Yte_predict = nn.predict(Xte_rows) # predict labels on the test images
# and now print the classification accuracy, which is the average number
# of examples that are correctly predicted (i.e. label matches)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )
```

Notice that 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. Of course, we've left out the meat of things - the actual classifier itself. Here is an implementation of a simple Nearest Neighbor classifier with the L1 distance that satisfies this template:

```
import numpy as np

class NearestNeighbor(object):
  def __init__(self):
    pass
```

```
def train(self, X, y):
  """ X is N x D where each row is an example. Y is 1-dimension of size N """
  # the nearest neighbor classifier simply remembers all the training data
  self.Xtr = X
  self.ytr = y

def predict(self, X):
  """ X is N x D where each row is an example we wish to predict label for """
  num_test = X.shape[0]
  # lets make sure that the output type matches the input type
  Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

  # loop over all test rows
  for i in range(num_test):
    # find the nearest training image to the i'th test image
    # using the L1 distance (sum of absolute value differences)
    distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
    min_index = np.argmin(distances) # get the index with smallest distance
    Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

  return Ypred
```

If you ran this code, 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 (see the leaderboard of a recent Kaggle competition on CIFAR-10).

The choice of distance. 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:

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

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. In numpy, using the code from above we would need to only replace a single line of code. The line that computes the distances:

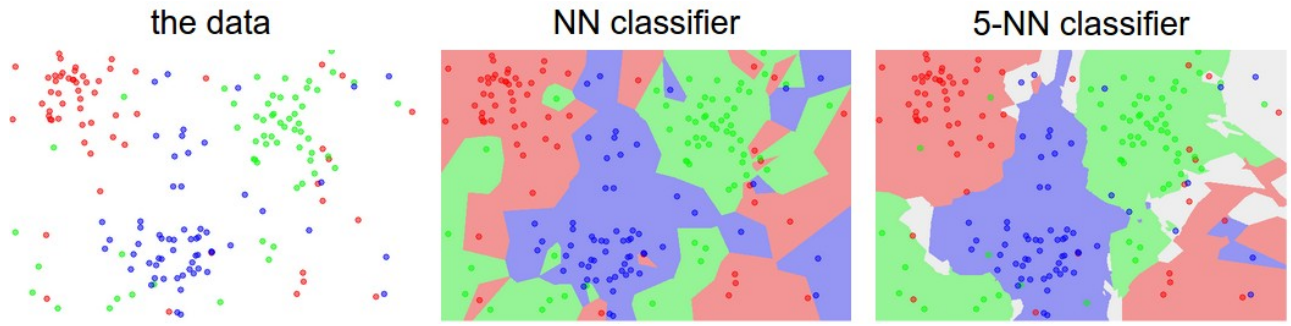
```
distances = np.sqrt(np.sum(np.square(self.Xtr - X[i,:]), axis = 1))
```

Note that I included the `np.sqrt` call above, but 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).

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.

k - Nearest Neighbor Classifier

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:



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.

Validation sets for 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 (we will see much more discussion surrounding generalization later in the class).

> *Evaluate on the test set only a single time, at the very end.*

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 49,000 of the training images for training, and leave 1,000 aside for validation. This validation set is essentially used as a fake test set to tune the hyper-parameters.

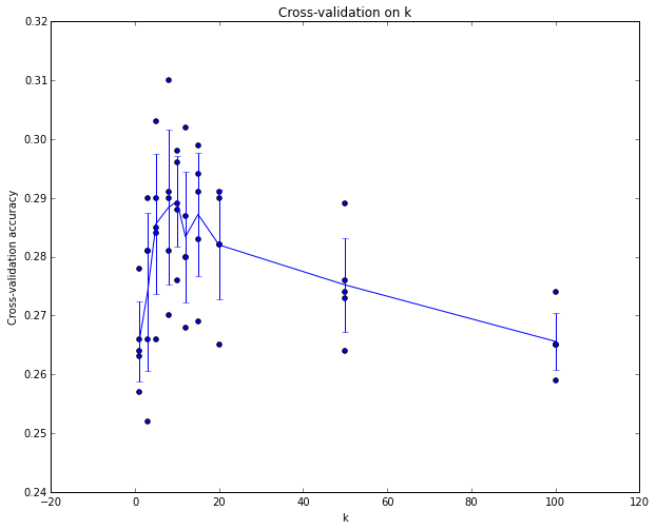
Here is what this might look like in the case of CIFAR-10:

```
# assume we have Xtr_rows, Ytr, Xte_rows, Yte as before
# recall Xtr_rows is 50,000 x 3072 matrix
Xval_rows = Xtr_rows[:1000, :] # take first 1000 for validation
Yval = Ytr[:1000]
Xtr_rows = Xtr_rows[1000:, :] # keep last 49,000 for train
Ytr = Ytr[1000:]

# find hyperparameters that work best on the validation set
validation_accuracies = []
for k in [1, 3, 5, 10, 20, 50, 100]:

  # use a particular value of k and evaluation on validation data
  nn = NearestNeighbor()
  nn.train(Xtr_rows, Ytr)
  # here we assume a modified NearestNeighbor class that can take a k as input
  Yval_predict = nn.predict(Xval_rows, k = k)
  acc = np.mean(Yval_predict == Yval)
  print 'accuracy: %f' % (acc,)

  # keep track of what works on the validation set
  validation_accuracies.append((k, acc))
```

By the end of this procedure, we could plot a graph that shows which values of *k* work best. We would then stick with this value and evaluate once on the actual test set.

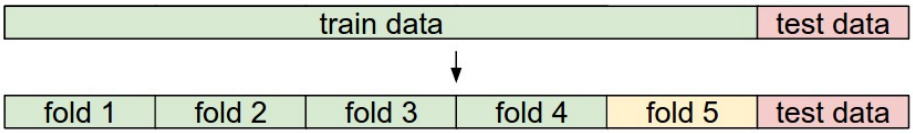
> *Split your training set into training set and a validation set. Use validation set to tune all hyperparameters. At the end run a single time on the test set and report performance.*

Cross-validation. 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.



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.

In practice. 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.



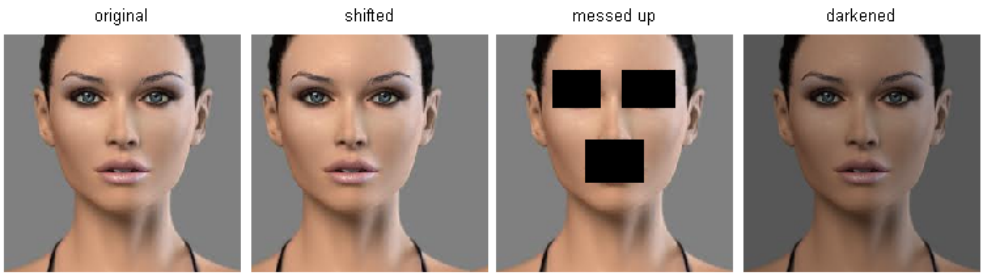
Common data splits. A training and test set is given. The training set is split into folds (for example 5 folds here). The folds 1-4 become the training set. One fold (e.g. fold 5 here in yellow) is denoted as the Validation fold and is used to tune the hyperparameters. Cross-validation goes a step further 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).

Pros and Cons of Nearest Neighbor classifier.

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

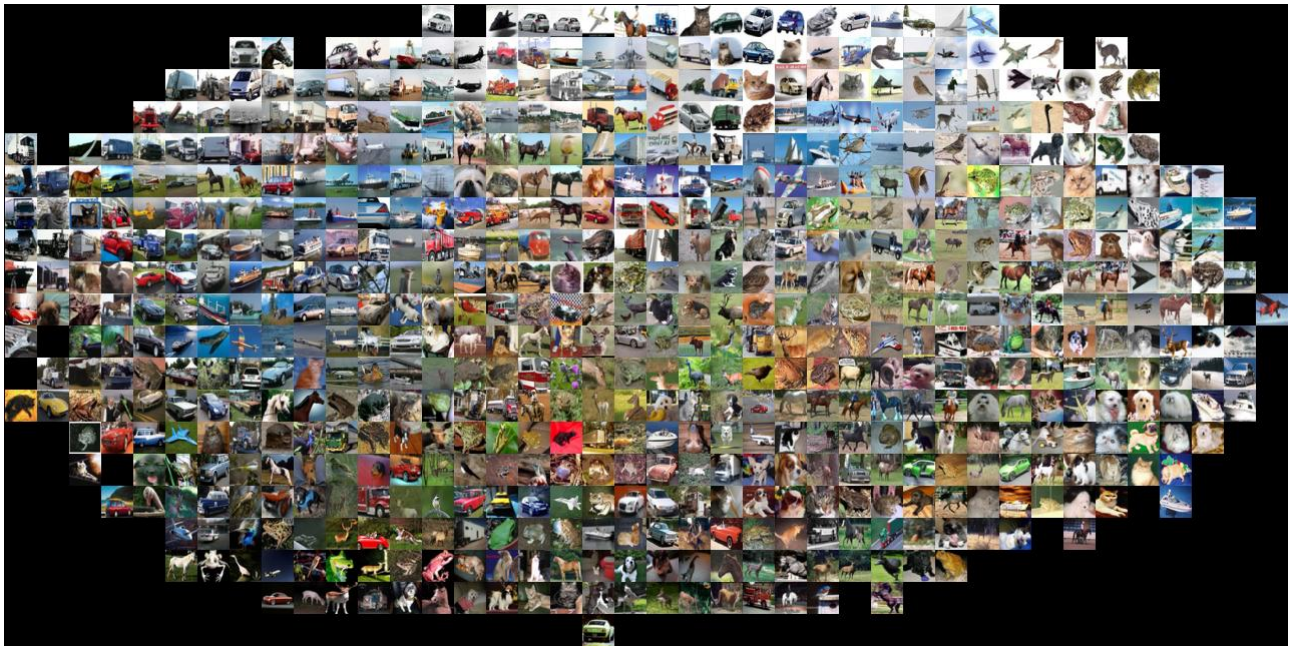
As an aside, the computational complexity of the Nearest Neighbor classifier is an active area of research, and several **Approximate Nearest Neighbor** (ANN) algorithms and libraries exist that can accelerate the nearest neighbor lookup in a dataset (e.g. FLANN). These algorithms allow one to trade 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.

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. The image below illustrates the point that the pixel-based L2 similarities we developed above are very different from perceptual similarities:



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.

Here is one more visualization to convince you that using pixel differences to compare images is inadequate. We can use a visualization technique called t-SNE 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 pixelwise distance we developed above:



CIFAR-10 images embedded in two dimensions with t-SNE. 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.

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.

Applying kNN in practice

If you wish to apply kNN in practice (hopefully not on images, or perhaps as only a baseline) proceed as follows:

1. Preprocess your data: Normalize the features in your data (e.g. one pixel in images) to have zero mean and unit variance. We will cover this in more detail in later sections, and chose not to cover data normalization in this section because pixels in images are usually homogeneous and do not exhibit widely different distributions, alleviating the need for data normalization.
2. If your data is very high-dimensional, consider using a dimensionality reduction technique such as PCA (wiki ref, CS229ref, blog ref), NCA (wiki ref, blog ref), or even Random Projections.
3. Split your training data randomly into train/val splits. As a rule of thumb, between 70-90% of your data usually goes to the train split. This setting depends on how many hyperparameters you have and how much of an influence you expect them to have. If there are many hyperparameters to estimate, you should err on the side of having larger validation set to estimate them effectively. If you are concerned about the size of your validation data, it is best to split the training data into folds and perform cross-validation. If you can afford the computational budget it is always safer to go with cross-validation (the more folds the better, but more expensive).

4. Train and evaluate the kNN classifier on the validation data (for all folds, if doing cross-validation) for many choices of **k** (e.g. the more the better) and across different distance types (L1 and L2 are good candidates)
5. If your kNN classifier is running too long, consider using an Approximate Nearest Neighbor library (e.g. FLANN) to accelerate the retrieval (at cost of some accuracy).
6. Take note of the hyperparameters that gave the best results. There is a question of whether you should use the full training set with the best hyperparameters, since the optimal hyperparameters might change if you were to fold the validation data into your training set (since the size of the data would be larger). In practice it is cleaner to not use the validation data in the final classifier and consider it to be *burned* on estimating the hyperparameters. Evaluate the best model on the test set. Report the test set accuracy and declare the result to be the performance of the kNN classifier on your data.

Linear Classification

In the last section we introduced the problem of Image Classification, which is the task of assigning a single label to an image from a fixed set of categories. Morever, we described the k-Nearest Neighbor (kNN) classifier which labels images by comparing them to (annotated) images from the training set. As we saw, kNN has a number of disadvantages:

- The classifier must *remember* all of the training data and store it for future comparisons with the test data. This is space inefficient because datasets may easily be gigabytes in size.
- Classifying a test image is expensive since it requires a comparison to all training images.

Overview. We are now going to develop a more powerful approach to image classification that we will eventually naturally extend to entire Neural Networks and Convolutional Neural Networks. The approach will have two major components: a **score function** that maps the raw data to class scores, and a **loss function** that quantifies the agreement between the predicted scores and the ground truth labels. We will then cast this as an optimization problem in which we will minimize the loss function with respect to the parameters of the score function.

Parameterized mapping from images to label scores

The first component of this approach is to define the score function that maps the pixel values of an image to confidence scores for each class. We will develop the approach with a concrete example. As before, let's assume a training dataset of images $x_i \in R^D$, each associated with a label y_i. Here $i = 1 \ldots N$ and $y_i \in 1 \ldots K$. That is, we have **N** examples (each with a dimensionality **D**) and **K** distinct categories. For example, in CIFAR-10 we have a training set of **N** = 50,000 images, each with **D** = 32 x 32 x 3 = 3072 pixels, and **K** = 10, since there are 10 distinct classes (dog, cat, car, etc). We will now define the score function $f : R^D \mapsto R^K$ that maps the raw image pixels to class scores.

Linear classifier. In this module we will start out with arguably the simplest possible function, a linear mapping:

$$f(x_i, W, b) = Wx_i + b$$

In the above equation, we are assuming that the image x_i has all of its pixels flattened out to a single column vector of shape [D x 1]. The matrix **W** (of size [K x D]), and the vector **b** (of size [K x 1]) are the **parameters** of the function. In CIFAR-10, x_i contains all pixels in the i-th image flattened into a single [3072 x 1] column, **W** is [10 x 3072] and **b** is [10 x 1], so 3072 numbers come into the function (the raw pixel values) and 10 numbers come out (the class scores). The parameters in **W** are often called the **weights**, and **b** is called the **bias vector** because it influences the output scores, but without interacting with the actual data x_i. However, you will often hear people use the terms *weights* and *parameters* interchangeably.

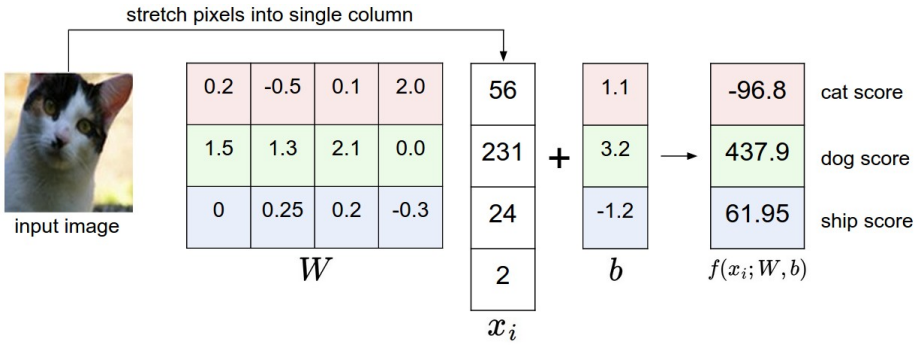
There are a few things to note:

- First, note that the single matrix multiplication Wx_i is effectively evaluating 10 separate classifiers in parallel (one for each class), where each classifier is a row of **W**.
- Notice also that we think of the input data (x_i, y_i) as given and fixed, but we have control over the setting of the parameters **W,b**. Our goal will be to set these in such way that the computed scores match the ground truth labels across the whole training set. We will go into much more detail about how this is done, but intuitively we wish that the correct class has a score that is higher than the scores of incorrect classes.
- An advantage of this approach is that the training data is used to learn the parameters **W,b**, but once the learning is complete we can discard the entire training set and only keep the learned parameters. That is because a new test image can be simply forwarded through the function and classified based on the computed scores.
- Lastly, note that classifying the test image involves a single matrix multiplication and addition, which is significantly faster than comparing a test image to all training images.

> *Foreshadowing: Convolutional Neural Networks will map image pixels to scores exactly as shown above, but the mapping (f) will be more complex and will contain more parameters.*

Interpreting a linear classifier

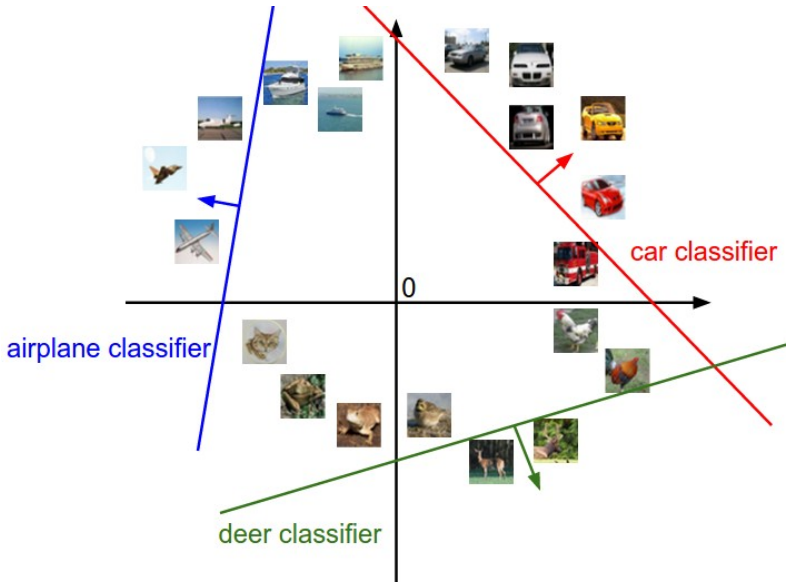
Notice that a linear classifier computes the score of a class as a weighted sum of all of its pixel values across all 3 of its color channels. Depending on precisely what values we set for these weights, the function has the capacity to like or dislike (depending on the sign of each weight) certain colors at certain positions in the image. For instance, you can imagine that the "ship" class might be more likely if there is a lot of blue on the sides of an image (which could likely correspond to water). You might expect that the "ship" classifier would then have a lot of positive weights across its blue channel weights (presence of blue increases score of ship), and negative weights in the red/green channels (presence of red/green decreases the score of ship).



An example of mapping an image to class scores. For the sake of visualization, we assume the image only has 4 pixels (4 monochrome pixels, we are not considering color channels in this example for brevity), and that we have 3 classes (red (cat), green (dog), blue (ship) class). (Clarification: in particular, the colors here simply indicate 3 classes and are not related to the RGB channels.) We stretch the image pixels into a column and perform matrix multiplication to get the scores for each class. Note that this particular set of weights W is not good at all: the weights assign our cat image a very low cat score. In particular, this set of weights seems convinced that it's looking at a dog.

Analogy of images as high-dimensional points. Since the images are stretched into high-dimensional column vectors, we can interpret each image as a single point in this space (e.g. each image in CIFAR-10 is a point in 3072-dimensional space of 32x32x3 pixels). Analogously, the entire dataset is a (labeled) set of points.

Since we defined the score of each class as a weighted sum of all image pixels, each class score is a linear function over this space. We cannot visualize 3072-dimensional spaces, but if we imagine squashing all those dimensions into only two dimensions, then we can try to visualize what the classifier might be doing:



Cartoon representation of the image space, where each image is a single point, and three classifiers are visualized. Using the example of the car classifier (in red), the red line shows all points in the space that get a score of zero for the car class. The red arrow shows the direction of increase, so all points to the right of the red line have positive (and linearly increasing) scores, and all points to the left have a negative (and linearly decreasing) scores.

As we saw above, every row of W is a classifier for one of the classes. The geometric interpretation of these numbers is that as we change one of the rows of W, the corresponding line in the pixel space will rotate in different directions. The biases b, on the other hand, allow our classifiers to translate the lines. In particular, note that without the bias terms, plugging in $x_i = 0$ would always give score of zero regardless of the weights, so all lines would be forced to cross the origin.

Interpretation of linear classifiers as template matching. Another interpretation for the weights W is that each row of W corresponds to a *template* (or sometimes also called a *prototype*) for one of the classes. The score of each class for an image is then obtained by comparing each template with the image using an *inner product* (or *dot product*) one by one to find the one that "fits" best. With this terminology, the linear classifier is doing template matching, where the templates are learned. Another way to think of it is that we are still effectively doing Nearest Neighbor, but instead of having thousands of training images we are only using a single image per class (although we will learn it, and it does not necessarily have to be one of the images in the training set), and we use the (negative) inner product as the distance instead of the L1 or L2 distance.

Skipping ahead a bit: Example learned weights at the end of learning for CIFAR-10. Note that, for example, the ship template contains a lot of blue pixels as expected. This template will therefore give a high score once it is matched against images of ships on the ocean with an inner product.

Additionally, note that the horse template seems to contain a two-headed horse, which is due to both left and right facing horses in the dataset. The linear classifier *merges* these two modes of horses in the data into a single template. Similarly, the car classifier seems to have merged several modes into a single template which has to identify cars from all sides, and of all colors. In particular, this template ended up being red, which hints that there are more red cars in the CIFAR-10 dataset than of any other color. The linear classifier is too weak to properly account for different-colored cars, but as we will see later neural networks will allow us to perform this task. Looking ahead a bit, a neural network will be able to develop intermediate neurons in its hidden layers that could detect specific car types (e.g. green car facing left, blue car facing front, etc.), and neurons on the next layer could combine these into a more accurate car score through a weighted sum of the individual car detectors.

Bias trick. Before moving on we want to mention a common simplifying trick to representing the two parameters W, b as one. Recall that we defined the score function as:

$$f(x_i, W, b) = Wx_i + b$$

As we proceed through the material it is a little cumbersome to keep track of two sets of parameters (the biases b and weights W) separately. A commonly used trick is to combine the two sets of parameters into a single matrix that holds both of them by extending the vector x_i with one additional dimension that always holds the constant 1 - a default *bias dimension*. With the extra dimension, the new score function will simplify to a single matrix multiply:

$$f(x_i, W) = Wx_i$$

With our CIFAR-10 example, x_i is now [3073 x 1] instead of [3072 x 1] - (with the extra dimension holding the constant 1), and W is now [10 x 3073] instead of [10 x 3072]. The extra column that W now corresponds to the bias b. An illustration might help clarify:

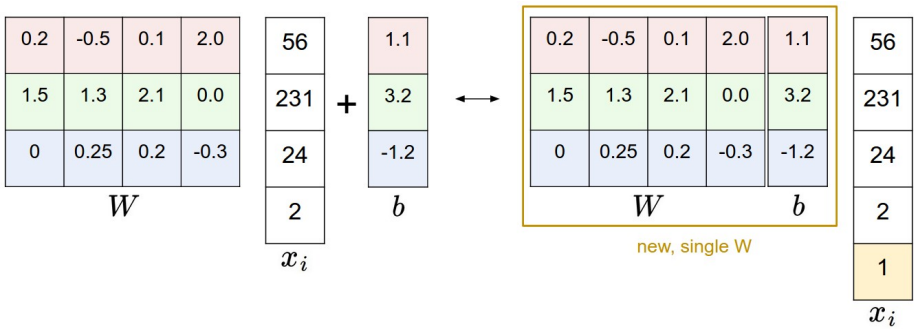


Illustration of the bias trick. Doing a matrix multiplication and then adding a bias vector (left) is equivalent to adding a bias dimension with a constant of 1 to all input vectors and extending the weight matrix by 1 column - a bias column (right). Thus, if we preprocess our data by appending ones to all vectors we only have to learn a single matrix of weights instead of two matrices that hold the weights and the biases.

Image data preprocessing. As a quick note, in the examples above we used the raw pixel values (which range from [0...255]). In Machine Learning, it is a very common practice to always perform normalization of your input features (in the case of images, every pixel is thought of as a feature). In particular, it is important to **center your data** by subtracting the mean from every feature. In the case of images, this corresponds to computing a *mean image* across the training images and subtracting it from every image to get images where the pixels range from approximately [-127 ... 127]. Further common preprocessing is to scale each input feature so that its values range from [-1, 1]. Of these, zero mean centering is arguably more important but we will have to wait for its justification until we understand the dynamics of gradient descent.