Example

Let's start with an example. Suppose we are charged with providing automated access control to a building. Before entering the building each person has to look into a camera so we can take a still image of their face. For our purposes it suffices just to decide based on the image whether the person can enter the building. It might be helpful to (try to) also identify each person but this might require type of information we do not have (e.g., names or whether any two face images correspond to the same person). We only have face images of people recorded while access control was still provided manually. As a result of this experience we have *labeled* images. An image is labeled *positive* if the person in question should gain entry and *negative* otherwise. To supplement the set of negatively labeled images (as we would expect only few cases of refused entries under normal circumstances) we can use any other face images of people who we do not expect to be permitted to enter the building. Images taken with similar camera-face orientation (e.g., from systems operational in other buildings) would be preferred. Our task then is to come up with a function – a *classifier* – that maps pixel images to binary (± 1) labels. And we only have the small set of labeled images (the *training set*) to constrain the function.

Let's make the task a bit more formal. We assume that each image (grayscale) is represented as a column vector $\mathbf{x}$ of dimension d. So, the pixel intensity values in the image, column by column, are concatenated into a single column vector. If the image has 100 by 100 pixels, then $d = 10000$. We assume that all the images are of the same size. Our classifier is a binary valued function $f : \mathcal{R}^d \to \{-1, 1\}$ chosen on the basis of the training set alone. For our task here we assume that the classifier knows nothing about images (or faces for that matter) beyond the labeled training set. So, for example, from the point of view of the classifier, the images could have been measurements of weight, height, etc. rather than pixel intensities. The classifier only has a set of n training vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ with binary ± 1 labels $y_1, \ldots, y_n$. This is the only information about the task that we can use to constraint what the function f should be.

What kind of solution would suffice?

Suppose now that we have $n = 50$ labeled pixel images that are 128 by 128, and the pixel intensities range from 0 to 255. It is therefore possible that we can find a single pixel, say pixel i, such that each of our n images have a distinct value for that pixel. We could then construct a simple binary function based on this single pixel that perfectly maps the training images to their labels. In other words, if x_{ti} refers to pixel i in the t^{th} training

image, and x'_i is the i^{th} pixel in any image $\mathbf{x}'$, then

$$f_i(\mathbf{x}') = \begin{cases} y_t, \text{ if } x_{ti} = x'_i \text{ for some } t = 1, \ldots, n \text{ (in this order)} \\ -1, \text{ otherwise} \end{cases} \quad (1)$$

would appear to solve the task. In fact, it is always possible to come up with such a "perfect" binary function if the training images are distinct (no two images have identical pixel intensities for all pixels). But do we expect such rules to be useful for images not in the training set? Even an image of the same person varies somewhat each time the image is taken (orientation is slightly different, lighting conditions may have changed, etc). These rules provide no sensible predictions for images that are not identical to those in the training set. The primary reason for why such trivial rules do not suffice is that our task is *not* to correctly classify the training images. Our task is to find a rule that works well for all new images we would encounter in the access control setting; the training set is merely a helpful source of information to find such a function. To put it a bit more formally, we would like to find classifiers that *generalize* well, i.e., classifiers whose performance on the training set is representative of how well it works for yet unseen images.

*What is model selection problem?

Model selection

So how can we find classifiers that generalize well? The key is to constrain the set of possible binary functions we can entertain. In other words, we would like to find a class of binary functions such that if a function in this class works well on the training set, it is also likely to work well on the unseen images. The "right" class of functions to consider cannot be too large in the sense of containing too many clearly different functions. Otherwise we are likely to find rules similar to the trivial ones that are close to perfect on the training set but do not generalize well. The class of function should not be too small either or we run the risk of not finding any functions in the class that work well even on the training set. If they don't work well on the training set, how could we expect them to work well on the new images? Finding the class of functions is a key problem in machine learning, also known as the *model selection* problem.

Linear classifiers through origin

Let's just fix the function class for now. Specifically, we will consider only a type of *linear classifiers*. These are thresholded linear mappings from images to labels. More formally, we only consider functions of the form

$$f(\mathbf{x}; \theta) = \text{sign}\big(\theta_1 x_1 + \ldots + \theta_d x_d\big) = \text{sign}\big(\theta^T \mathbf{x}\big) \quad (2)$$

Vector Dot Product:
We can calculate the sum of the multiplied elements of two vectors of the same length to give a scalar.This is called the dot product, named because of the dot operator used when describing the operation.
c = a . b
The operation can be used in machine learning to calculate the weighted sum of a vector.The dot product is calculated as follows:
a . b = (a1 * b1 + a2 * b2 + a3 * b3) => pythonda soyle yaziliyor.

where $\theta = [\theta_1, \ldots, \theta_d]^T$ is a column vector of real valued parameters. Different settings of the parameters give different functions in this class, i.e., functions whose value or *output* in $\{-1, 1\}$ could be different for some *input* images $\mathbf{x}$. Put another way, the functions in our class are parameterized by $\theta \in \mathcal{R}^d$.

We can also understand these linear classifiers geometrically. The classifier changes its prediction only when the argument to the sign function changes from positive to negative (or vice versa). Geometrically, in the space of image vectors, this transition corresponds to crossing the *decision boundary* where the argument is exactly zero: all $\mathbf{x}$ such that $\theta^T\mathbf{x} = 0$. The equation defines a plane in d-dimensions, a plane that goes through the origin since $\mathbf{x} = 0$ satisfies the equation. The parameter vector θ is normal (orthogonal) to this plane; this is clear since the plane is defined as all $\mathbf{x}$ for which $\theta^T\mathbf{x} = 0$. The θ vector as the normal to the plane also specifies the direction in the image space along which the value of $\theta^T\mathbf{x}$ would increase the most. Figure 1 below tries to illustrate these concepts.

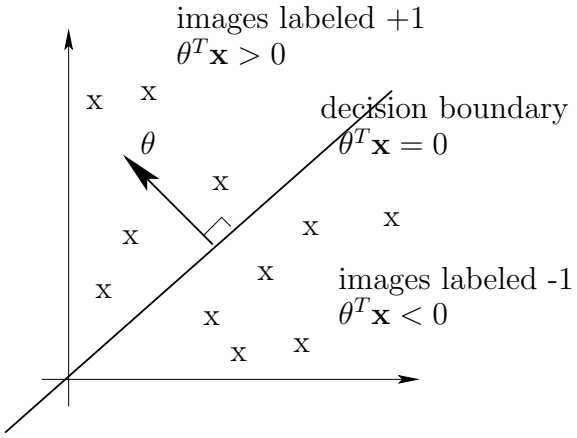

Figure 1: A linear classifier through origin.

Before moving on let's figure out whether we lost some useful properties of images as a result of restricting ourselves to linear classifiers? In fact, we did. Consider, for example, how nearby pixels in face images relate to each other (e.g., continuity of skin). This information is completely lost. The linear classifier is perfectly happy (i.e., its ability to classify images remains unchanged) if we get images where the pixel positions have been reordered provided that we apply the same transformation to all the images. This permutation of pixels merely reorders the terms in the argument to the sign function in Eq. (2). A linear classifier therefore does not have access to information about which pixels are close to each other in the image.

by using linear classifiers, did we lost any inf. about image
yes, pixel positions
but linear classifiers doesnot have access to information which pixels are close to each other in the image.

Learning algorithm: the perceptron

Now that we have chosen a function class (perhaps suboptimally) we still have to find a specific function in this class that works well on the training set. This is often referred to as the *estimation* problem. Let's be a bit more precise. We'd like to find a linear classifier that makes the fewest mistakes on the training set. In other words, we'd like to find θ that minimizes the *training error*

$$\hat{E}(\theta) = \frac{1}{n}\sum_{t=1}^{n}\left(1 - \delta(y_t, f(\mathbf{x}_t;\theta))\right) = \frac{1}{n}\sum_{t=1}^{n}\text{Loss}(y_t, f(\mathbf{x}_t;\theta)) \tag{3}$$

where $\delta(y, y') = 1$ if $y = y'$ and 0 otherwise. The training error merely counts the average number of training images where the function predicts a label different from the label provided for that image. More generally, we could compare our predictions to labels in terms of a loss function $\text{Loss}(y_t, f(\mathbf{x}_t;\theta))$. This is useful if errors of a particular kind are more costly than others (e.g., letting a person enter the building when they shouldn't). For simplicity, we use the *zero-one loss* that is 1 for mistakes and 0 otherwise.

What would be a reasonable algorithm for setting the parameters θ? Perhaps we can just incrementally adjust the parameters so as to correct any mistakes that the corresponding classifier makes. Such an algorithm would seem to reduce the training error that counts the mistakes. Perhaps the simplest algorithm of this type is the *perceptron* update rule. We consider each training image one by one, cycling through all the images, and adjust the parameters according to

$$\theta' \leftarrow \theta + y_t\mathbf{x}_t \;\text{ if }\; y_t \neq f(\mathbf{x}_t;\theta) \tag{4}$$

In other words, the parameters (classifier) is changed only if we make a mistake. These updates tend to correct mistakes. To see this, note that when we make a mistake the sign of $\theta^T\mathbf{x}_t$ disagrees with y_t and the product $y_t\theta^T\mathbf{x}_t$ is negative; the product is positive for correctly classified images. Suppose we make a mistake on $\mathbf{x}_t$. Then the updated parameters are given by $\theta' = \theta + y^t\mathbf{x}_t$, written here in a vector form. If we consider classifying the same image $\mathbf{x}_t$ after the update, then

$$y_t\theta'^T\mathbf{x}^t = y_t(\theta + y_t\mathbf{x}_t)^T\mathbf{x}_t = y_t\theta^T\mathbf{x}_t + y_t^2\mathbf{x}_t^T\mathbf{x}_t = y_t\theta^T\mathbf{x}_t + \|\mathbf{x}_t\|^2 \tag{5}$$

In other words, the value of $y_t\theta^T\mathbf{x}_t$ increases as a result of the update (becomes more positive). If we consider the same image repeatedly, then we will necessarily change the parameters such that the image is classified correctly, i.e., the value of $y_t\theta^T\mathbf{x}_t$ becomes positive. Mistakes on other images may steer the parameters in different directions so it may not be clear that the algorithm converges to something useful if we repeatedly cycle through the training images.

Analysis of the perceptron algorithm

The perceptron algorithm ceases to update the parameters only when all the training images are classified correctly (no mistakes, no updates). So, if the training images are possible to classify correctly with a linear classifier, will the perceptron algorithm find such a classifier? Yes, it does, and it will converge to such a classifier in a finite number of updates (mistakes). We'll show this in lecture 2.