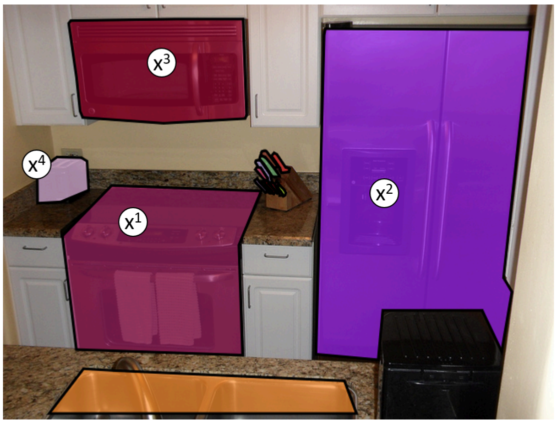
7.1 Discriminant functions

In the previous chapter, we explored how to extract a feature vector $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$ to describe an image object or *region*. In this chapter, our goal is **to classify the object as belonging to one of M object classes or categories**, according to said feature vector $\mathbf{x}$. In other words, we pursuit the design of a mapping between features and categories $\mathbf{x}^n \rightarrow C$ where $C \in \{C_1, \dots, C_M\}$ (see Fig 1.).



Scenario: A computer vision system in charge of classifying the objects within kitchens. The considered categories are: $C_1 = \text{fridge}$, $C_2 = \text{bowl}$, $C_3 = \text{microwave}$, $C_4 = \text{oven}$, $C_5 = \text{toaster}$, $C_6 = \text{spoon}$.

Results of object classification:

$$x^1 = [x_1^1, x_2^1, \dots, x_n^1]^T \rightarrow C^1 = C_4$$
$$x^2 = [x_1^2, x_2^2, \dots, x_n^2]^T \rightarrow C^2 = C_1$$
$$x^3 = [x_1^3, x_2^3, \dots, x_n^3]^T \rightarrow C^3 = C_3$$
$$x^4 = [x_1^4, x_2^4, \dots, x_n^4]^T \rightarrow C^4 = C_5$$

Fig 1. Example of object classification within the COCO dataset.

For that, the space defined by the features in $\mathbf{x}$, also referred as **feature space**, is divided into prediction regions. The image below shows an example of those regions in a scenario with categories $C = \{C_1, C_2, C_3\}$, so $M = 3$, and features $\mathbf{x} = [x_1, x_2]^T$:

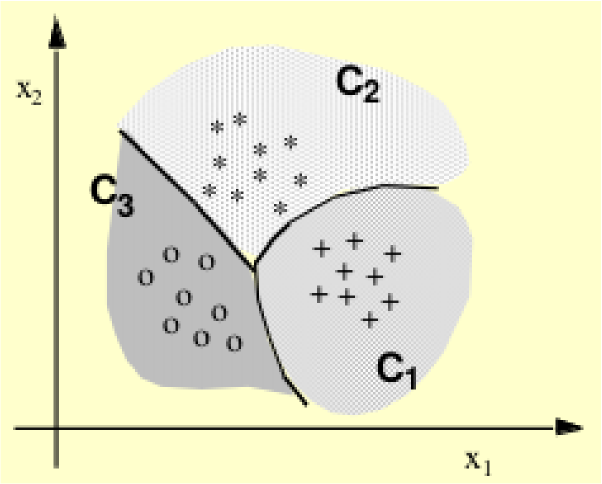


Fig 2. Example of a feature space with three regions.

Two main steps are needed in order to build an object recognition system and work with it:

- A **training (design) phase**, where sample vectors of known objects are used to learn the classifier (supervised learning).
- A **prediction (online) phase**, where the image objects are classified as belonging to one of the classes based on the learned prediction models.

Such prediction models come in two flavors:

- **Statistical classifiers:**
  - It is assumed that the feature vectors $\mathbf{x}$ of the classes C follow a **statistical distribution**.
  - The parameters of such distribution need to be **learned from known objects**.
  - Examples: Naïve Bayessian Classifier, Logistic Regression, Conditional Random Fields.
- **Non-statistical classifiers:**
  - No assumption is made on the statistical distribution of the feature vector.
  - The coefficient of deterministic discriminant functions are learned.
  - Examples: Support Vector Machine, Perceptron, AdaBoost, Neural Networks, Genetic algorithms.

This notebook focuses on non-statistical classifiers. Concretely it covers:

- Linear discriminant functions (section 7.1.1).
- The concept of separability (section 7.1.2).
- Generalized discriminant functions (section 7.1.3).

All these methos are under the Machine Learning (ML) umbrela. If you want to give a try to Deep Learning (DL) ones, you can do it in notebook *7.5 Deep Learning methods for object detection*!

Problem context - Traffic sign recognition

Before self-driving vehicles can truly operate autonomously, they would need to identify traffic signs like "speed limit", "school ahead" or "turn ahead". This problem is called **Traffic Sign Recognition (TSR)** and is faced by companies like Tesla or Google (Waymo), and now by *AliquindoiCars*, a startup located in the Andalusia Technology Park in Málaga.

TSR is part of the features collectively called *Advanced Driver Assistance Systems (ADAS)*. It uses image processing techniques to detect the traffic signs. The detection methods used can be generally divided into color based, shape based, or those using low level features.

Given your growing experience in computer vision, *AliquindoiCars* contacted you asking for a TSR technique to be integrated into a self-driving car.

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['figure.figsize'] = (8.0, 8.0)

images_path = './images/'
```

The data

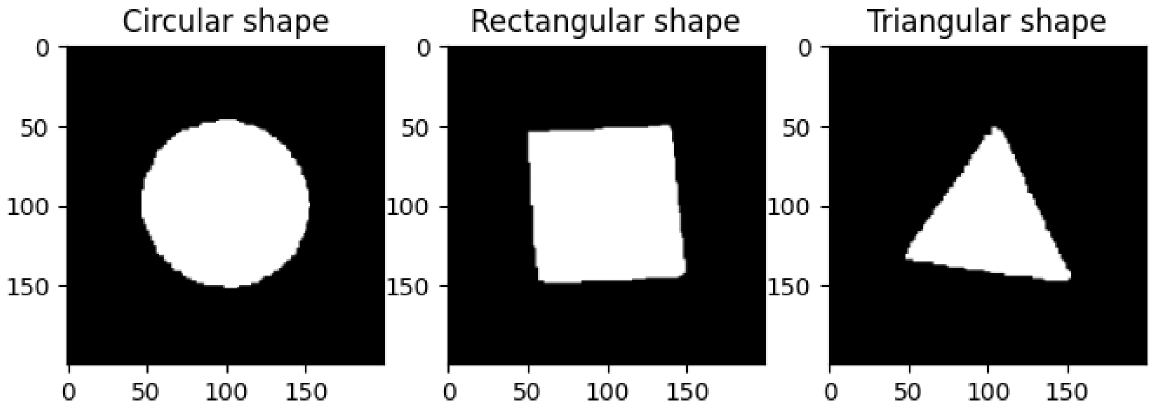
Since most traffic signs in Spain (and in the rest of the world) have a rectangular, circular or triangular shape, they provided you 3 folders containing 20 images each;

- `./images/circles/{0-19}.png`
- `./images/squares/{0-19}.png`
- `./images/triangles/{0-19}.png`

These folders contain segmented and binarized images of traffic signs captured by a camera in a car. **Our task is to design a recognition system able to distinguish each type of shape**.

Let's take a look at some of these images:

In [2]:
```
# Load images
img_circle = cv2.imread(images_path + "circles/0.png", 0)
img_square = cv2.imread(images_path + "squares/0.png", 0)
img_triangle = cv2.imread(images_path + "triangles/0.png", 0)

# Show them!
plt.subplot(131)
plt.imshow(img_circle,cmap='gray');
plt.title('Circular shape')

plt.subplot(132)
plt.imshow(img_square,cmap='gray');
plt.title('Rectangular shape')
plt.subplot(133)
plt.imshow(img_triangle,cmap='gray');
plt.title('Triangular shape');
```



Computing feature vectors

Prior to describe the methods that will be used to perform recognition, let's build the feature vectors they are going to use. For this, and given your experience, you are going to use the Hu Moments, a 7-size feature vector that looks like $\mathbf{x} = [v_1, v_2, v_3, v_4, v_5, v_6, v_7]^T$.

ASSIGNMENT 1: Computing the features

Your first task is to obtain a $20x7$ matrix for each type of shape, where rows index images and columns Hu moments. In this way, the 5^{th} row in the matrix characterizing rectangular signs would contain the Hu moments computed from the 5^{th} image with that shape.

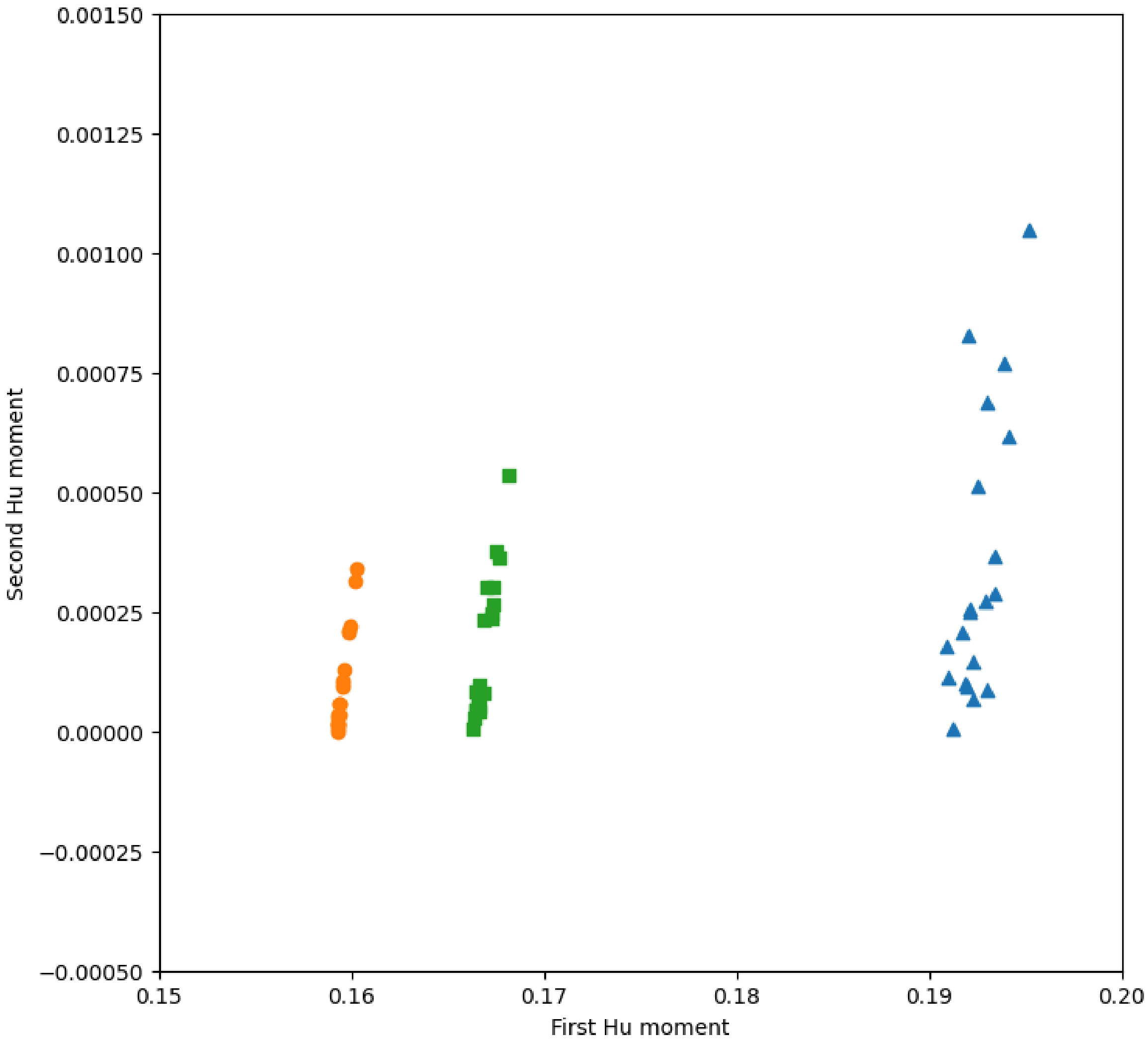
In order to get a first-hand impression of how these features are distributed in the feature space for the different shapes, plot the results using `plt.scatter()` (show only the firsts 2 Hu moments).

In [3]:

```
# Assignment 1
def image_moments(region):
    """ Compute moments of the external contour in a binary image.

        Args:
            region: Binary image

        Returns:
            moments: dictionary containing all moments of the region
    """

    # Get external contour
    contours,_ = cv2.findContours(region,cv2.RETR_EXTERNAL ,cv2.CHAIN_APPROX_NON
    cnt = contours[0]

    # Compute moments
    moments = cv2.moments(cnt)

    return moments
```

In [4]:

```python
# Compute Hu moments for circle shape
hu_circles = np.zeros((20,7))

for i in range(20):
    img = cv2.imread(images_path + "circles/" + str(i) + ".png", 0)
    moments = image_moments(img)
    hu_circles[i,:] = cv2.HuMoments(moments).flatten()

# Compute Hu moments for square shape
hu_squares = np.zeros((20,7))

for i in range(20):
    img = cv2.imread(images_path + "squares/" + str(i) + ".png", 0)
    moments = image_moments(img)
    hu_squares[i,:] = cv2.HuMoments(moments).flatten()

# Compute Hu moments hor triangle shape
hu_triangles = np.zeros((20,7))

for i in range(20):
    img = cv2.imread(images_path + "triangles/" + str(i) + ".png", 0)
    moments = image_moments(img)
    hu_triangles[i,:] = cv2.HuMoments(moments).flatten()

# Define plot axis
plt.axis([0.15, 0.2, -0.0005, 0.0015])

# Plot firsts two Hu moments
plt.xlabel("First Hu moment")
plt.ylabel("Second Hu moment")
plt.scatter(hu_triangles[:,0],hu_triangles[:,1],marker="^")
plt.scatter(hu_circles[:,0],hu_circles[:,1], marker="o")
plt.scatter(hu_squares[:,0],hu_squares[:,1], marker="s");
```



Interlude

You must save the computed Hu moments of the segmented data, since we are going to need them in the next notebook. For saving numPy matrices, you can use `np.save()` (data is saved at `./data/` by default).

In [5]:

```
np.save("./data/hu_circles.npy",hu_circles)
np.save("./data/hu_triangles.npy",hu_triangles)
np.save("./data/hu_squares.npy",hu_squares)
```

7.1.1 Linear discriminant functions

Taking a look at the scatter plot resultant from the previous assignment, we can see how the three types of signs could be segmented using 2 lines. For example:

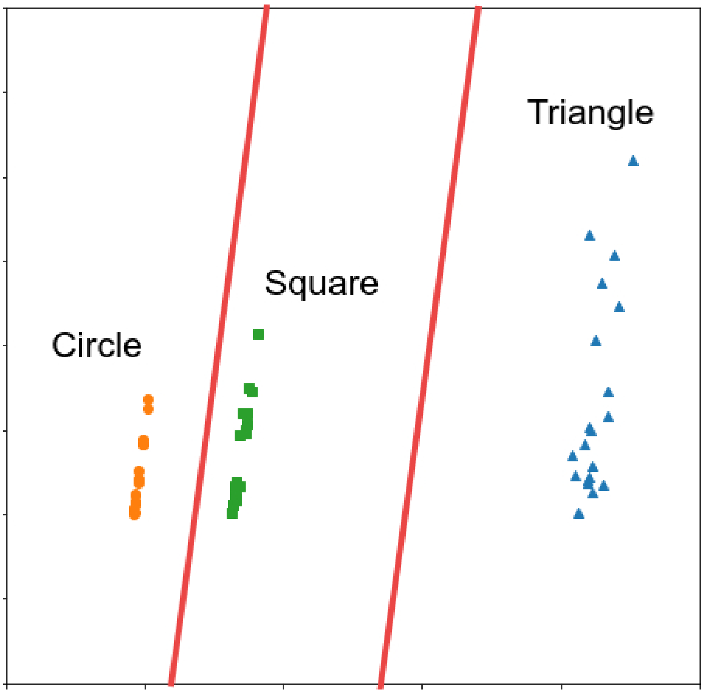


Fig 3. First two moments of the binarized traffic signs and a possible division into regions.

And that's what **linear discriminant functions** do!

Let's start by describing how such lines could be represented. As we know, a way to define a 2D line is:

$$0 = Ax + By + C$$

Using another notation:

$0 = w_1x_1 + w_2x_2 + w_3 \backslash [3pt]$ Note that we can generalize this 2D line to any dimension hyperplane:$\backslash [5pt]$

$$0 = w_1x_1 + w_2x_2 + \ldots + w_nx_n + w_{n+1}1 \;\; = \;\; \sum_{i=1}^{n+1} w_ix_i$$

$$= \;\; w^T \cdot x$$

In this way, we can divide the feature space for any dimension (e.g. n = 7 for Hu moments).

Working with two classes

If we are considering just two classes $C = \{C_1, C_2\}$ (*e.g.* an object in a kitchen could be a spoon or a fork), we can define a linear discriminant function as:

$$d(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} \qquad \text{(dot product of both vectors)}$$

Formally, the vector w is called **weight vector**, and is computed automatically from a set of data called **training data** using techniques like Logistic Regression, Support Vector

Machines, or Perceptron.

In this way, when we have a new object characterized by $\mathbf{x}'$ to classify, $d(\mathbf{x}')$ is computed and:

- If $d(\mathbf{x}') \geq 0$ the new object is assigned C_1, and
- if $d(\mathbf{x}') < 0$ the object is assigned to C_2.

We say that the linear discriminant function $d(\mathbf{x}')$ works as a **decision boundary** between both classes. The following figure illustrates this (blue dots represent characterized objects belonging to C_1, while red ones those belonging to C_2).

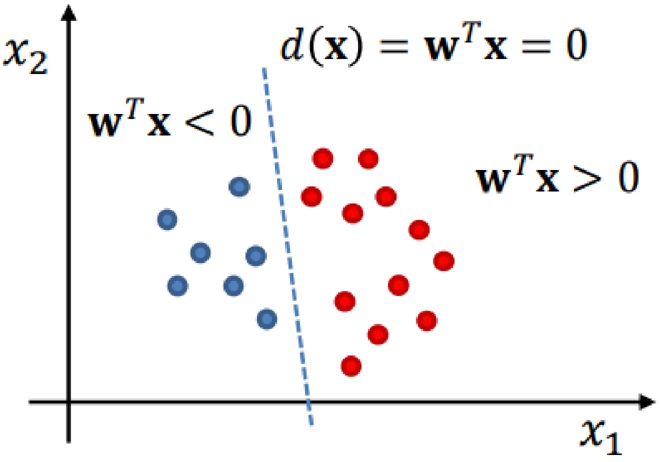

x_2
$d(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = 0$
$\mathbf{w}^T\mathbf{x} < 0$
$\mathbf{w}^T\mathbf{x} > 0$
x_1

Fig 3. A decision boundary defined by a decision function that divides the feature space into two regions.

ASSIGNMENT 2. Classifying samples

Let's play a bit with a toy example of a linear discriminant function. Consider that the learned weights during the training phase of the classifier are $\mathbf{w} = [1, 0, -3]^T$, so the linear discriminant function looks like $d(\mathbf{x}) = 1x_1 + 0x_2 - 3$.

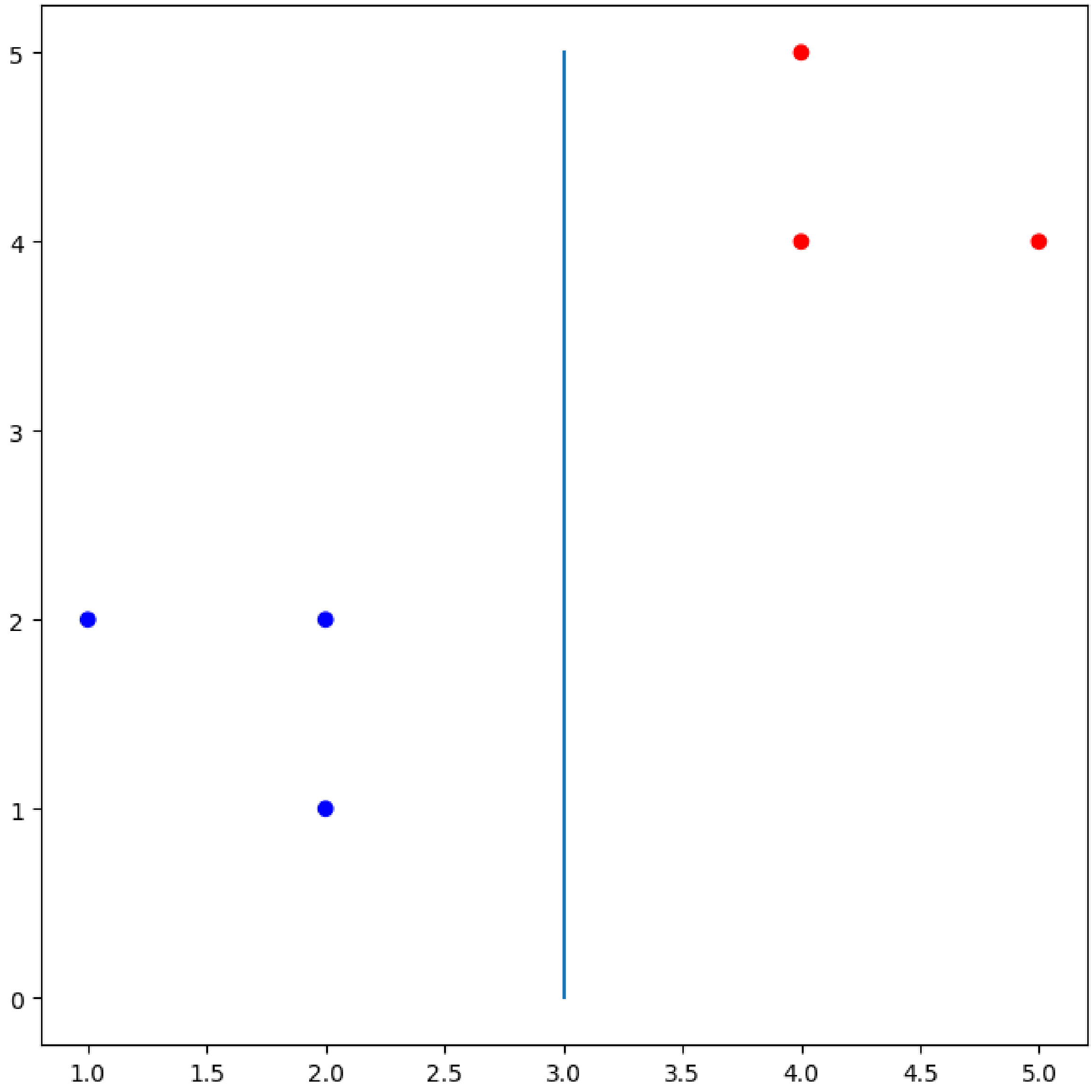
Your task is to:

- classify some 2D data using the previous discriminant function (for that, use a combination of `np.dot()` and `np.sign()`), and
- plot them with different colors depending on their assigned class using `plt.scatter()` and its `c` argument.

Tip: how to color a data point using its class.

In [6]:
```
# Define weight vector
w = np.array([[1],[0],[-3]])

# Define data matrix
data = np.array([[2,2,1],[1,2,1],[2,1,1],[5,4,1],[4,5,1],[4,4,1]])

# Compute class for each data
classification = np.dot(data,w)
```

```
class_ = np.sign(classification)

# Plot classified data
plt.scatter(data[:,0],data[:,1],c=class_.flatten(), cmap="bwr")

# Plot discriminant line
x = np.linspace(3, 3, 1000)
y = np.linspace(0, 5, 1000)
plt.plot(x,y);
```



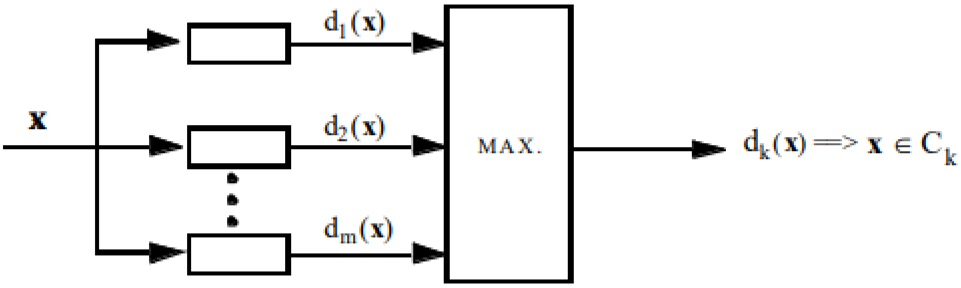
Scenarios with N-classes

The previous idea can be generalized to an arbitrary number of classes, defining in that case hyper-planes in a n-dimensional space:

$$d_i(\mathbf{x}) = w_1^i x_1 + w_2^i x_2 + \ldots + w_n^i x_n + w_{n+1}^i \quad = \quad \mathbf{w^i}^T \cdot \mathbf{x}$$

In this case, the object is assigned to the category which discriminant function returns the highest value. That is:

$$\text{IF} \quad d_i(\mathbf{x}) > d_j(\mathbf{x}) \quad \forall i \neq j \quad \text{THEN} \ \mathbf{x} \in C_i$$



When more than 2 classes are considered, **the discriminant functions are no longer the decision boundaries between classes**. However, the decision boundary between classes C_i and C_j can be computed as:

$$d_{ij}(\mathbf{x}) = d_i(\mathbf{x}) - d_j(\mathbf{x}) = (\mathbf{w}_i^T - \mathbf{w}_j^T) \cdot \mathbf{x} = \mathbf{w}_{ij}^T \cdot \mathbf{x}$$

The following image shows an example with 3 classes (notice that it also exists a decision boundary between classes 1 and 3, $d_{13}(\mathbf{x})$, but it has been omitted for clarity).

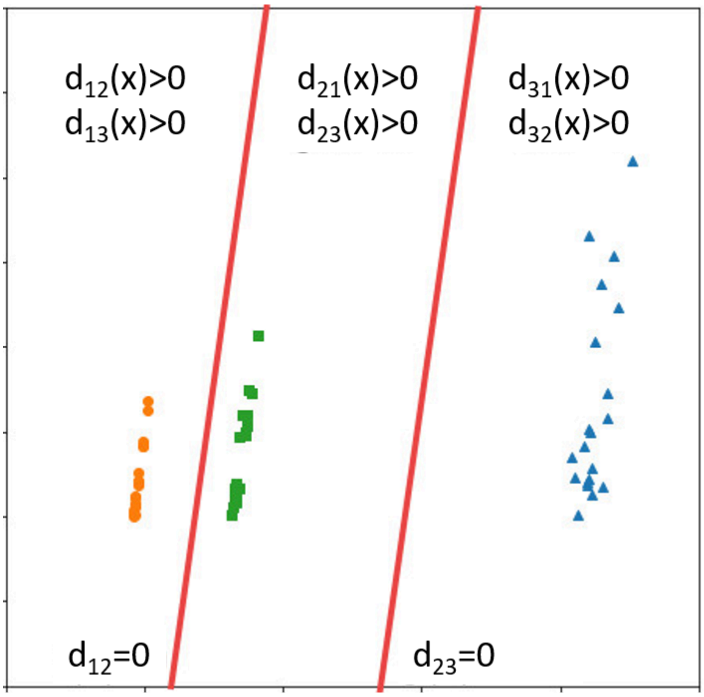


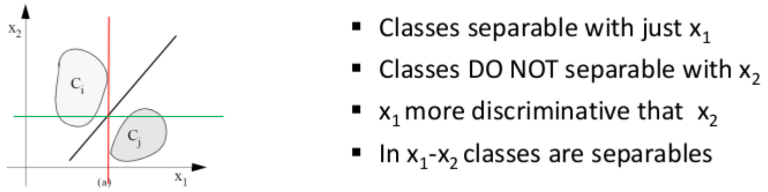
Fig 4. Decision boundaries between 3 categories.

7.1.2 Separability

An important concept regarding object recognition is such of **separability**. In this way, two sets of points defining two classes may be separable or not in a given dimension (e.g. x_1, x_2, etc.).

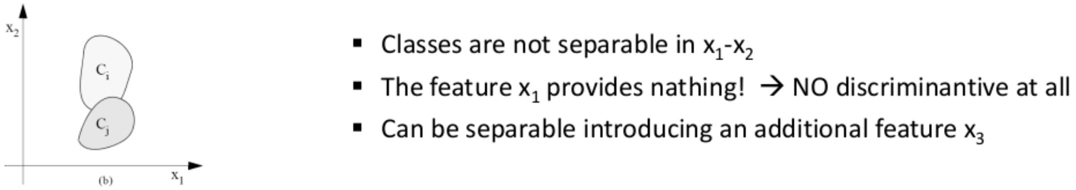
Two sets are **linearly separable** if it exists at least one line in the plane that leaves all the points belonging to one class on one side of the line and all those belonging to the second class on the other side. This idea immediately generalizes to higher-dimensional Euclidean spaces if the line is replaced by a hyperplane.

Some illustrative examples of this:



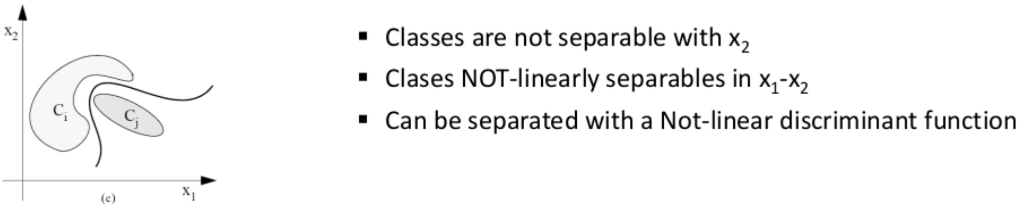
- Classes separable with just x_1
- Classes DO NOT separable with x_2
- x_1 more discriminative that x_2
- In x_1-x_2 classes are separables

- As you can see, those classes are separable. Also, if only dimension x_1 is used, we can separate the classes as well. This is not recomended because there isn't a clear separation using only x_1, and x_2 is also **discriminative**, that is, it also provides valuable information for separating both classes.



- Classes are not separable in x_1-x_2
- The feature x_1 provides nathing! → NO discriminantive at all
- Can be separable introducing an additional feature x_3

- In this example the classes are not separable as they overlap in both dimensions. Additionally, we can see that x_1 is not discriminative as it doesn't provide any information for separating those classes. The addition of a third dimension/feature could fix this!



- Classes are not separable with x_2
- Clases NOT-linearly separables in x_1-x_2
- Can be separated with a Not-linear discriminant function

- This is an example of a non-linear separable problem. The classes can't be separated using a line, but a **generalized discriminant function** could do it. We will see this later in this notebook.

Back to sign shape recognition

Back to our traffic sign shape recognition problem, **we are trying to separate** the usual shapes of traffic signs using the Hu moments. As a first approximation to this problem, we plotted the first and second Hu moments (x_1 and x_2) in order to analyze them. You should have obtained something like this:

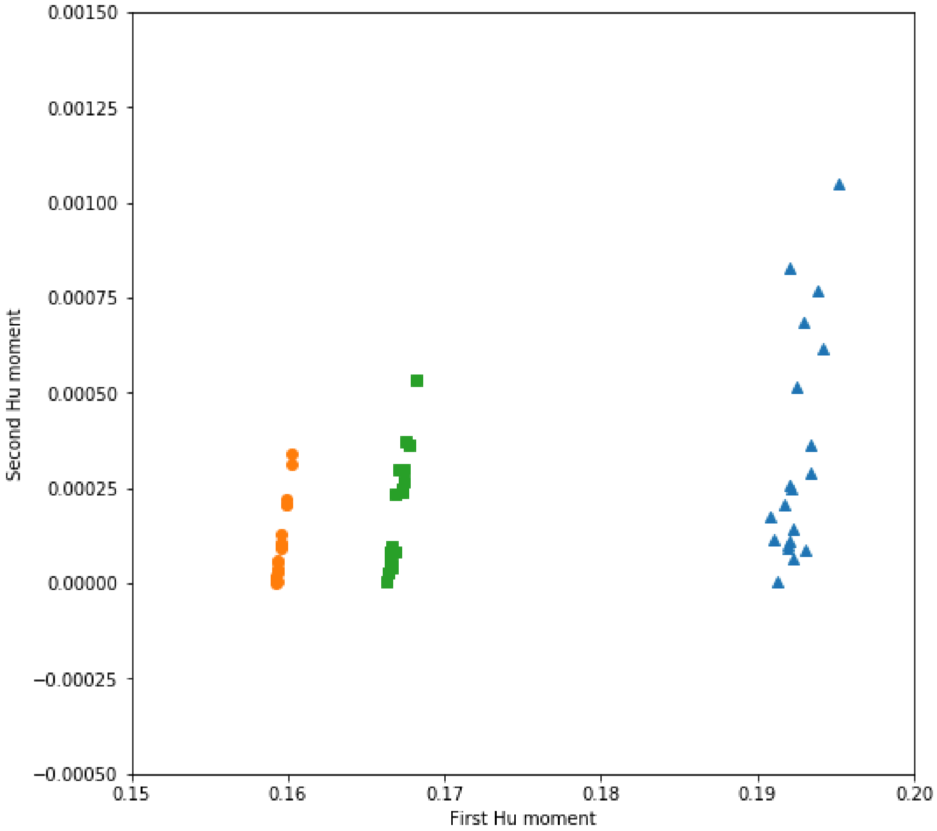


Fig 5. Result of plotting the two first Hu moments computed from the images in our train data.

Thinking about it (1)

Now you are in a good position to answer these questions:

- Is this problem linearly separable?

  Yes, it is, as each class can be separated using lines.

- Are the classes separable using only the first Hu moment (x_1)?

  Yes, the first Hu moment is discriminative enough to separate all 3 classes.

- Are the classes separable using only the second Hu moment (x_2)?

  No, the second Hu moment is not discriminative enough to separate them, it does not provide information at all.

- Which dimension is more discriminative?

  Definitely the x1 dimension, that is, the one belonging to the first Hu moment.

- Would be *AliquindoiCars* guys happy if we propose a solution for shape classification based on linear discriminant functions?

  Yes, they would. These classes (the different sign shapes) are linearly separable thanks to the first Hu moment.

7.1.3 Generalized discriminant function

As we saw in the previous section, some classes are not separable using a linear discriminant function. In those cases, we may need a **linear basis function model**, also called **generalized discriminant function**, which permit us to use non-linear functions to separate classes (notice that, despite of this, the problem may still not be separable).

The idea behind these discriminant functions is to transform the $\mathbf{x}$ space into a $\mathbf{x'} = \mathbf{f}(\mathbf{x})$ space ($dim(x) < dim(x')$):

$$d(x) = w_1\ f_1(x) + w_2\ f_2(x) + \ldots + w_k\ f_k(x) + w_{k+1} = \sum_{i=1}^{k+1} w_i\ \underbrace{f_i(x)}_{\text{New space}} = \mathbf{w}^T \cdot \mathbf{f}(\mathbf{x})$$

$$d(x) = w_1\ x'_1 + w_2\ x'_2 + \ldots + w_k\ x'_k + w_{k+1} = \sum_{i=1}^{k+1} w_i\ x'_i = w^T \cdot x' = d'(\mathbf{x}')$$

Thereby:

$$\left.\begin{array}{r}\mathbf{x}' = \mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}) \quad f_2(\mathbf{x}) \quad \cdots \quad f_k(\mathbf{x}) \quad 1]^T \\ \mathbf{w} = [w_1 \quad w_2 \quad \cdots \quad w_k \quad w_{k+1}]^T\end{array}\right\} \ d(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} = \mathbf{w}^T \cdot \mathbf{f}(\mathbf{x})$$

being $\mathbf{f}(\mathbf{x})$ the basis functions!

ASSIGNMENT 3: Transforming between spaces

Let's practice this transformation between spaces. **What to do?** You have to complete the method `transform_space()`, which transforms a set of 2D points into a new space defined by:

$$x' = [x_1^2,\ x_1 x_2,\ x_2^2,\ x_1,\ x_2,\ 1]$$

The input set of points is a matrix with shape $(n\_data, 2)$, and the output is another matrix with shape $(n\_data, 6)$, being n_data the number of points in the set.

In [7]:

```python
# Assignment 3
def transform_space(data_points):
    """ Compute moments of the external contour in a binary image.
    
        Args:
            data_points: Input matrix containing a set of 2D points
            
        Returns:
            fx: set of points transformed to a quadratic space
    """
    
    n_data = data_points.shape[0]
    
    fx = np.zeros((n_data,6))
    
    for i in range(n_data):
        x1 = data_points[i,0]
        x2 = data_points[i,1]

        fx[i,:] = [x1**2, x1*x2, x2**2, x1, x2, 1]

    return fx
```

You can use next code to **test if the results are correct**:

```
In [8]: data = np.array([[2,3],[1,2],[2,7],[2,4],[1,5],[2,4]])

        x = transform_space(data)

        print(x)
```

```
[[ 4.   6.   9.   2.   3.   1.]
 [ 1.   2.   4.   1.   2.   1.]
 [ 4.  14.  49.   2.   7.   1.]
 [ 4.   8.  16.   2.   4.   1.]
 [ 1.   5.  25.   1.   5.   1.]
 [ 4.   8.  16.   2.   4.   1.]]
```

Expected output:

```
[[ 4.   6.   9.   2.   3.   1.]
 [ 1.   2.   4.   1.   2.   1.]
 [ 4.  14.  49.   2.   7.   1.]
 [ 4.   8.  16.   2.   4.   1.]
 [ 1.   5.  25.   1.   5.   1.]
 [ 4.   8.  16.   2.   4.   1.]]
```

ASSIGNMENT 4: Classifying in the transformed space

Finally, we will see a toy example of a non-linear discriminant function in action. Concretely, we will consider a quadratic function, $d(x) = x_1^2 - 5x_2$ (so $k = 5$), in a two classes problem with $\mathbf{x} = [x_1 \;\; x_2]$. Thereby, the vector of features in the new space is:

$$\mathbf{x}' = [x_1^{'}, \ldots, x_6^{'}] = [x_1^2 \;\; x_1x_2 \;\; x_2^2 \;\; x_1 \;\; x_2 \;\; 1]^T$$

so:

- $x_1^2 = f_1(\mathbf{x}) = x_1'$
- $x_1x_2 = f_2(\mathbf{x}) = x_2'$
- and so on.

and the vector of weights turns out to be:

$$\mathbf{w} = [1, 0, 0, 0, -5, 0].$$

As for the new discriminant function we get:

$$d(\mathbf{x}) = w_1 \cdot x_1' + w_5 \cdot x_5'$$

Your task is to classify some 2D data using the previous discriminant function and plot them coloring each point with a color depending on its class. For that, you can rely on the `np.dot()` , `np.sign()`), and `plt.scatter()` functions previously used in the second assignment.

Tip: Remember to transform the data space before applying the dot product.

Expected results:

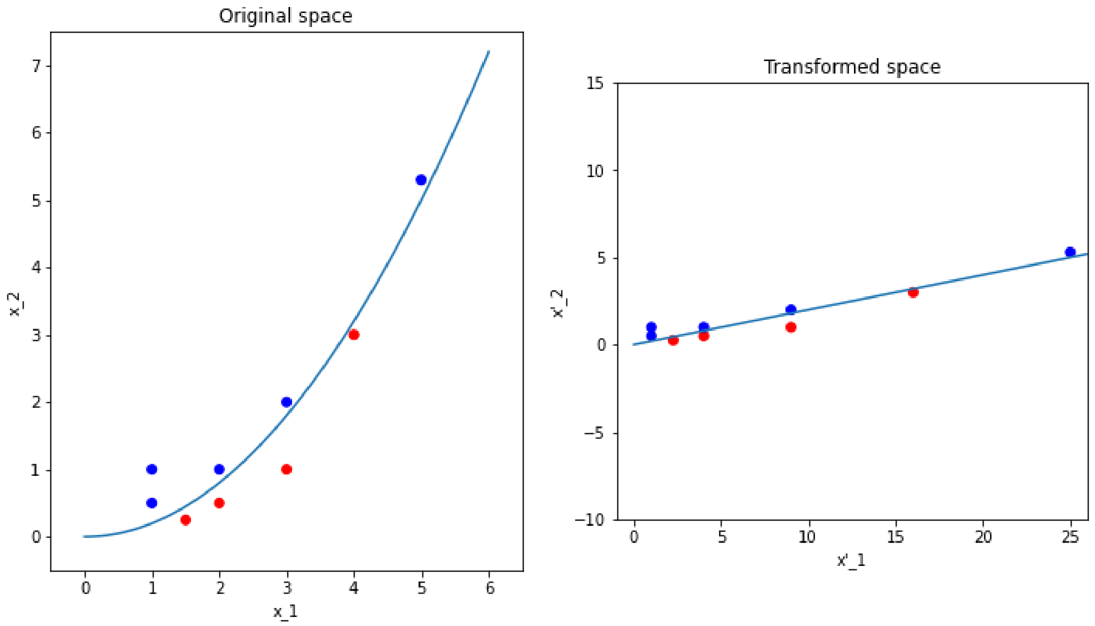
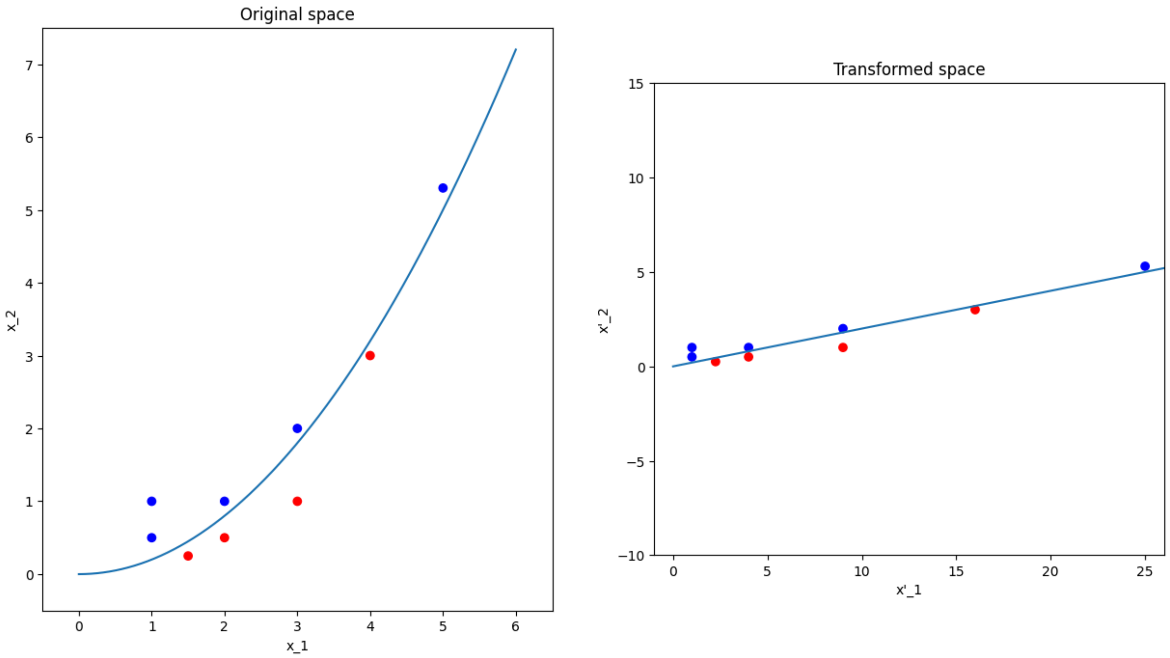


Fig 6. Classification of a number of samples in both, the original space and the transformed one.

In [9]:

```
# Assignment 4

# Define weight vector
w = np.array([[1],[0],[0],[0],[-5],[0]])

# Define data matrix
data = np.array([[2,0.5],[1.5,0.25],[3,1],[1,1],[1,0.5],[2,1],[5,5.3],[4,3],[3,2

# Trasform from the x space to a f(x) space
transformed_data = transform_space(data)

# Compute class for each data
classification = np.dot(transformed_data,w)
class_ = np.sign(classification)

# Plot classified data
matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)
ax = plt.subplot(121)
ax.set_aspect('equal')
plt.title('Original space')
plt.xlabel("x_1")
plt.ylabel("x_2")
plt.axis([-0.5, 6.5, -0.5, 7.5])
plt.scatter(data[:,0],data[:,1],c=class_.flatten(), cmap="bwr")

# Plot discriminant line
x = np.linspace(0, 6, 1000)
y = (w[0]*(x**2) + w[5])/-w[4]
```

```
plt.plot(x,y);

# Or equivalently plotting the data in the transformed space,
# where the descision boundary defined by the discrimiant function defines a lin
ax = plt.subplot(122)
ax.set_aspect('equal')
plt.title('Transformed space')
plt.xlabel("x'_1")
plt.ylabel("x'_2")
plt.axis([-1, 26, -10, 15])
plt.scatter(transformed_data[:,0],transformed_data[:,4],c=class_.flatten(), cmap

# Plot discrimant line
x = np.linspace(0, 27, 1000)
y = (w[0]*(x) + w[5])/-w[4]
plt.plot(x,y);
```



Conclusion

Well done! This was a mostly theoretical notebook, but it described the basis of object classification. You have learned so far:

- how linear and non-linear discriminant functions work,
- to classify objects in a scenario with an arbitrary number of belonging classes using multiple discriminant functions, and
- to check if a feature space allows for the definition of discriminative classifiers looking at the separability between classes.

In the following notebooks in this chapter we will see how to automatically retrieve the weight vector from a set o training data. Exciting, isn't?

7.4 Multilayer Perceptron for Classification

A Multilayer Perceptron (MLP) is a type of artificial neural network that consists of an input layer, one or more hidden layers, and an output layer. It is a powerful model used for supervised learning tasks, such as classification and regression.

Each neuron in the MLP applies a nonlinear activation function to a weighted sum of its inputs, allowing the network to learn complex patterns in the data. During **training**, by using backpropagation, the MLP adjusts the weights of connections in the network to minimize the error between predicted and actual values. During **inference**, the MLP is faced with new input data, producing an inference result.

MLPs are particularly effective for solving classification problems where the relationship between inputs and outputs is non-linear. In this notebook, we will use an MLP to classify iris flowers into three different species, and breast cancer cases as either malignant or benign, based on a set of features. For that, we will first explore the data we are going to work with (section 7.4.1), then the **Perceptron model** (the building block of many neural networks, section 7.4.2), and finally the **Multilayer Perceptron** (section 7.4.3).

This will be doing in the context of the scikit-learn library, an open source software for predictive analysis.

Problem Context: Iris and Breast Cancer Classification

In this notebook, you will have the option to explore two different classification tasks using machine learning:

1. Iris Classification:

The **Iris dataset** is a classic dataset used in machine learning and statistics. It contains 150 samples from three different species of Iris flowers: Setosa, Versicolor, and Virginica. Each sample has four features:

- Sepal length
- Sepal width
- Petal length
- Petal width

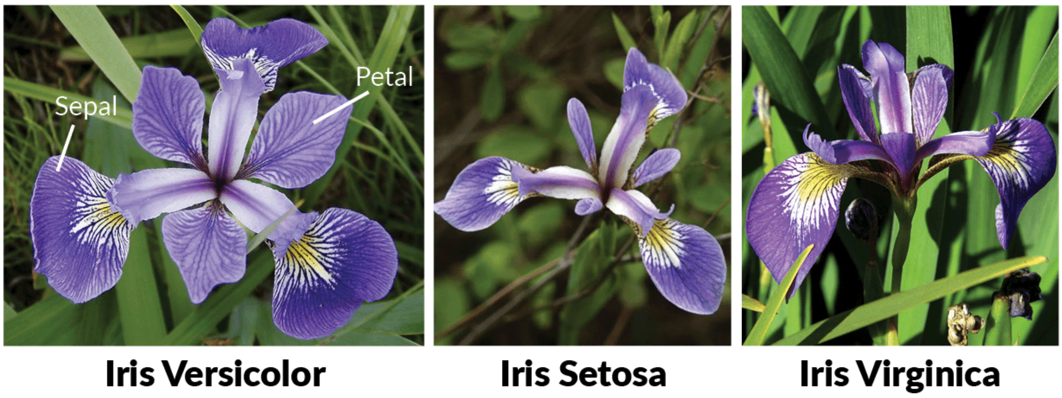
The task is to build a classifier that can predict the species of an Iris flower based on these four features. The target classes are:

- **0**: Setosa
- **1**: Versicolor

- **2**: Virginica

This is a **multiclass classification problem** where the goal is to distinguish between the three flower species. An interesting fact is that one class is linearly separable from the other 2, but the latter are not linearly separable from each other.

For your reference, here an example of each specie:



Iris Versicolor Iris Setosa Iris Virginica

Image from *Machine Learning in R for beginners*.

2. Breast Cancer Classification:

The **Breast Cancer dataset** is focused on medical diagnosis, aiming to classify whether a breast tumor is malignant (cancerous) or benign (non-cancerous) based on the features derived from a digitized image of a fine needle aspirate of a breast mass. This dataset includes 30 features that describe the characteristics of the cell nuclei, such as radius, texture, and smoothness.

The target classes in this dataset are:

- **0**: Malignant
- **1**: Benign

This is a **binary classification problem** where the goal is to correctly classify tumors as malignant or benign based on the input features.

Why these datasets?

Because they were or could be obtained using computer vision techniques. In both cases, the regions of interest could be segmentend and then characterized through a number of features.

We encourage you to take a look at their description in the scikit-learn page. They are part of the **Toy examples** collection, which can be accessed here. On the other hand, there are also larger datasets, listed in the **Real world datasets** collection available here.

The world wide web is full of datasets, there are platforms like kaggle that are a good entry point to this world.

Your Task:

- You will use a **Perceptron** (a single-layer neural network) and a **Multilayer Perceptron (MLP)** (a deep neural network with hidden layers) to classify both datasets.
- You can select which dataset you'd like to work with first and explore the performance of both models on these two datasets.

In [1]:

```
# Step 1: Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.datasets import load_wine
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_sc
```

7.4.1 Loading and analyzing the datasets

Let's load a dataset and take a look at its content. We will start working with the Iris dataset and we recommend you to complete the notebook with this one. Once completed, you can change to the Breast cancer classification or any other. Up to you!

In [2]:

```
# Load the dataset

# Iris dataset
iris_data = load_iris()
X = iris_data.data  # Features
y = iris_data.target  # Target labels
feature_names = iris_data.feature_names

# Breast cancer dataset
# breast_cancer_data = load_breast_cancer()
# X = breast_cancer_data.data  # Features (30-dimensional feature vectors)
# y = breast_cancer_data.target  # Target labels (0: malignant, 1: benign)
# feature_names = breast_cancer_data.feature_names

# wine_data = load_wine()
# X = wine_data.data  # Features (13-dimensional feature vectors)
# y = wine_data.target  # Target labels (0, 1, or 2, representing different wine
# feature_names = wine_data.feature_names

# Display basic information about the dataset
print(f"Number of samples: {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
print(f"Feature names: {feature_names}")
print(f"Target classes: {np.unique(y)}")

# Show the first 5 samples and their corresponding labels
print("First 5 samples (feature vectors):")
print(X[:5])
print("Corresponding labels:")
print(y[:5])
```

```
Number of samples: 150
Number of features: 4
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'pe
tal width (cm)']
Target classes: [0 1 2]
First 5 samples (feature vectors):
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
Corresponding labels:
[0 0 0 0 0]
```

ASSIGNMENT 1: Feature scaling

First of all, let's start by investigating the range of the different features of the dataset.

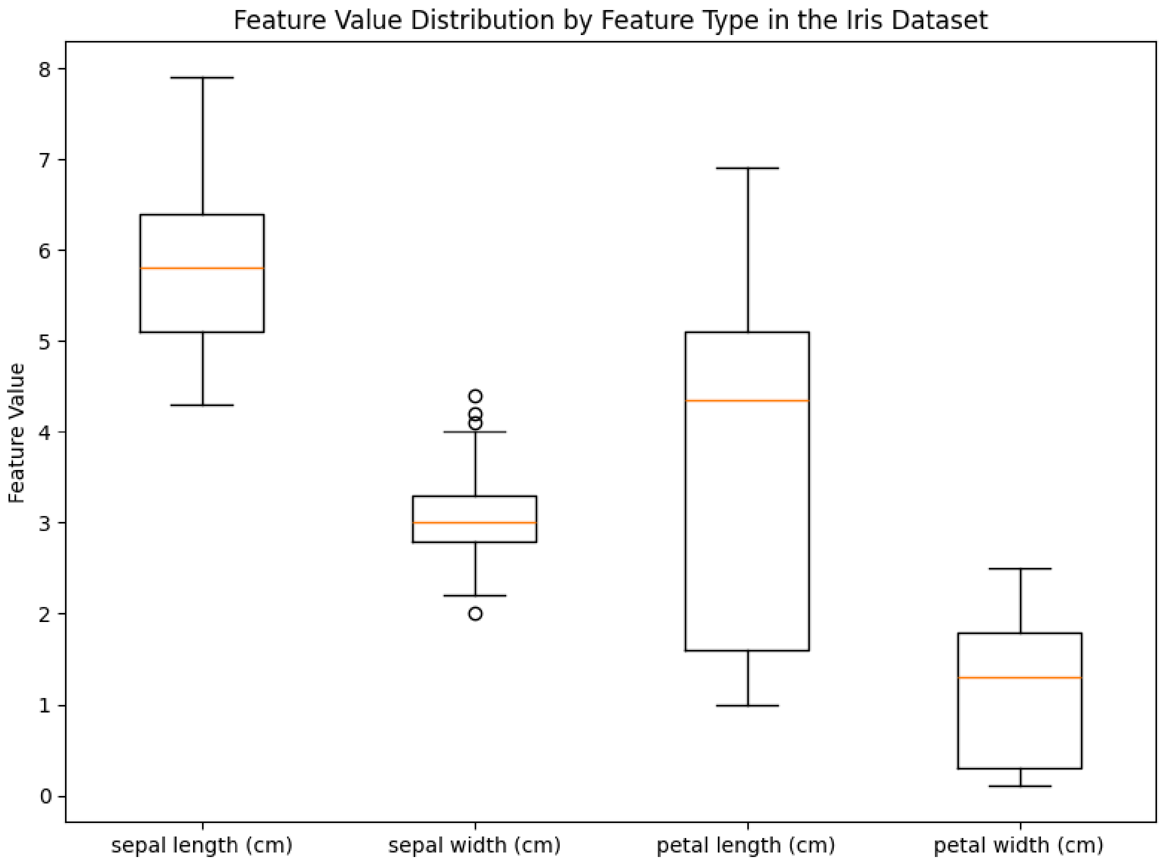
You are tasked to:

- Compute the minimum and maximum value of each feature.
- Represent their dispersion using a boxplot.
- Think about the obtained results.

In [3]:

```
# Display the minimum and maximum values of each feature
print("Feature minimums:", X.min(axis=0))
print("Feature maximums:", X.max(axis=0))

# Create a box plot for each feature
plt.figure(figsize=(8, 6))

# Create a box plot, each feature on the x-axis
plt.boxplot(X, tick_labels=feature_names)

# Add labels and title
plt.title('Feature Value Distribution by Feature Type in the Iris Dataset')
plt.ylabel('Feature Value')

# Show the plot
plt.tight_layout()
plt.show()
```

```
Feature minimums: [4.3 2.  1.  0.1]
Feature maximums: [7.9 4.4 6.9 2.5]
```



Expected output (with Iris dataset):

```
Feature minimums: [4.3 2.  1.  0.1]
Feature maximums: [7.9 4.4 6.9 2.5]
```

The Perceptron and MLP models, as many other machine learning techniques, face problems if the input data is **unscaled**. As you have experienced, input features in real-world datasets often vary widely in terms of their value ranges. These differences in scale cause problems because the learning process in neural networks relies heavily on gradient-based optimization algorithms, such as stochastic gradient descent (SGD).

If the features have different ranges, the model may struggle to balance the influence of each feature, causing several issues:

- Unequal updates to weights.
- Slow convergence.
- Vanishing or exploiding gradients.

To address this, we perform a **preprocessing step** to **scale** the input data so that all features have the same range **(more info here)**. Two common methods are **standarization**, where each feature is rescaled to have a mean of 0 and a standard deviation of 1, and **normalization**, where each feature is rescaled to fit within a specific range, typically $[0, 1]$.

We will play with standarization, which computes the new feature values as:

$$x' = \frac{x - \mu}{\sigma}$$

where:

- x is the original feature value,
- μ is the mean of the feature,
- σ is the standard deviation of the feature.

You **are tasked to** complete the following code so:

- It splits the dataset into training and testing data with `train_test_split()` (docs),
- Then normalizes the features' values using an instance of the `StandardScaler` class (docs). First fit the scaler with the training features, and then transform the testing one as well.
- Finally, it shows the feature values of the first 5 samples in the training data scaled. Complete it!

In [4]:

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.3, random_state=

# Standardize the features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Display the first 5 rows of the scaled data to check
print("First 5 rows of the scaled training data:")
print(X_train_scaled[:5])
```

```
First 5 rows of the scaled training data:
[[-0.4134164  -1.46200287 -0.09951105 -0.32339776]
 [ 0.55122187 -0.50256349  0.71770262  0.35303182]
 [ 0.67180165  0.21701605  0.95119225  0.75888956]
 [ 0.91296121 -0.02284379  0.30909579  0.2177459 ]
 [ 1.63643991  1.41631528  1.30142668  1.70589097]]
```

Expected output (with Iris dataset):

```
First 5 rows of the scaled training data:
[[-0.4134164  -1.46200287 -0.09951105 -0.32339776]
 [ 0.55122187 -0.50256349  0.71770262  0.35303182]
 [ 0.67180165  0.21701605  0.95119225  0.75888956]
 [ 0.91296121 -0.02284379  0.30909579  0.2177459 ]
 [ 1.63643991  1.41631528  1.30142668  1.70589097]]
```

7.4.2 Perceptron: The Building Block of Neural Networks

A **Perceptron** is the simplest form of a neural network and can be seen as a single neuron. It is a linear classifier that makes predictions by applying a linear combination of inputs and weights, followed by a threshold (or step) function to produce a **binary output**, so it is applied to two-class problems.

The Math Behind the Perceptron

For a given input vector of features $\tilde{\mathbf{x}} = [x_1, x_2, \dots, x_n, 1]^T$ and a corresponding weight vector $\mathbf{w} = [w_1, w_2, \dots, w_n, w_{n+1}]^T$, the perceptron computes a weighted sum of the inputs, called **pre-activation**:

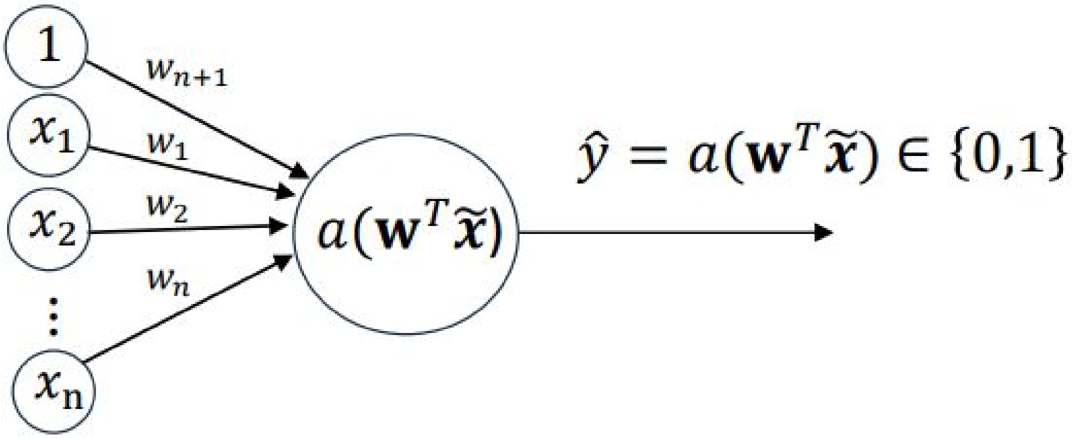
$$z = \sum_{i=1}^{n+1} w_i \tilde{x}_i = \mathbf{w}^T \tilde{\mathbf{x}}$$

You can also find in the literature definitions where the last weight, w_{n+1}, is left out and denoted b. At any case, it plays the role of a **bias term**, which allows the decision boundary to be shifted.

Once the weighted sum z is computed, the Perceptron applies an activation function $a(z)$ to determine the output:

$$\hat{y} = a(\mathbf{w}^T \tilde{\mathbf{x}}) \in \{0, 1\}$$

Visually:


1
x_1
x_2
x_n
w_{n+1}
w_1
w_2
w_n
$a(\mathbf{w}^T\tilde{x})$
$\hat{y} = a(\mathbf{w}^T\tilde{x}) \in \{0,1\}$

The traditional activation function for a Perceptron is the **step function**, which outputs:

$$\hat{y} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Learning

During training, the Perceptron updates its weights using the following rule:

$$w_i \leftarrow w_i + \Delta w_i$$

Where the weight update Δw_i is calculated as:

$$\Delta w_i = \eta \cdot e_i \cdot x_i = \eta \cdot (y - \hat{y}) \cdot x_i$$

Here:

- y is the true label.
- $\hat{y}$ is the predicted label.

- η is the learning rate, a scalar that controls how much the weights are adjusted with each step.

If the Perceptron makes an incorrect prediction (i.e., $y \neq \hat{y}$), the weights are adjusted in the direction that reduces the prediction error.

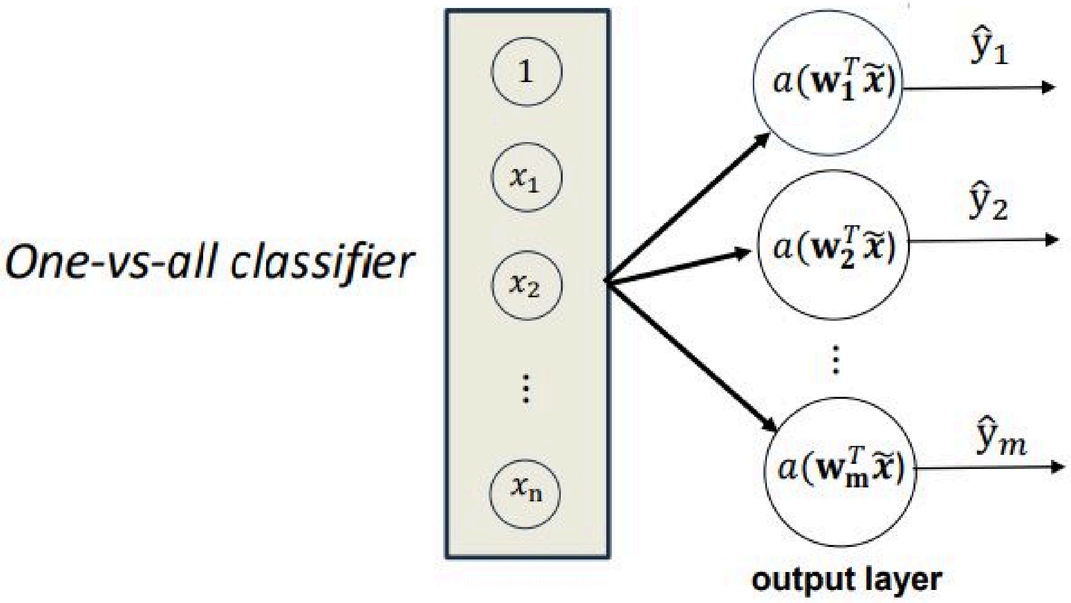
Extension to Multiclass problems

In scenario wehere there are more than two classes to classify a sample into, the perceptron can still be used to do the work. The idea here is to train one perceptron for each class, while keeping the same input layer. Notice that the activation functions are still linear!

The classification criteria in this case is as follows:

$$\hat{y} = \begin{cases} C_j & \text{if } \hat{y}_j = 1 \text{ and } \hat{y}_k = 0 \, \forall k \neq j \\ \text{undetermined} & \text{otherwise} \end{cases}$$

Visually:


One-vs-all classifier
1
x_1
x_2
⋮
x_n
$a(\mathbf{w}_1^T\tilde{\mathbf{x}})$
$a(\mathbf{w}_2^T\tilde{\mathbf{x}})$
⋮
$a(\mathbf{w}_m^T\tilde{\mathbf{x}})$
$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_m$
output layer

ASSIGNMENT 2: Testing Perceptron

Once we have the data ready, and we have reviewed the principles behind the Perceptron, it's time to train a model and check its performance.

Scikit-learn provides the `Perceptron` class (docs), which permits you to create instances of this model. Its constructor has many parameters, but perhaps the more relevant at this point are:

- `max_iter (int, default=1000)` : The maximum number of passes over the training data (aka epochs) to train the model.
- `tol (float or None, default=1e-3)` : The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

- `random_state (int, RandomState instance or None, default=0)` : Pass an int for reproducible output across multiple function calls. More info here.

Then, the method `fit()` permits you to fit the model using training data, while `predict()` permits you to do inference with new data, in this case `X_test_scaled` .

Complete the following code to create the model object and train it. To check that your results are right, do it with a maximun number of iterations of 1000, a tolerance of $1e-3$, and a random state of 42. Take a look at the shape of the model coeficients, which stores the fitted weights.

In [5]:

```python
# Initialize the Perceptron model (single-layer neural network)
perceptron_model = Perceptron(max_iter=1000,
                              tol=1e-3,
                              random_state=42)

# Train the Perceptron model on the scaled training data
perceptron_model.fit(X_train_scaled, y_train)

print('Model shape: ',perceptron_model.coef_.shape)
print('Intercepts: ',perceptron_model.intercept_.shape) # Weights associated to
```

```
Model shape:  (3, 4)
Intercepts:  (3,)
```

Now it's time to check the model's performance. For that:

- Compute the predictions for the flowers described in `X_test_scaled` .
- Measure the performance with the following metrics:
  - Accuracy, with `accuracy_score()` (docs),
  - The popular confusion matrix, with `confusion_matrix()` (docs),
  - And print a classification report with `classification_report()` (docs).

In [6]:

```python
# Make predictions using the trained Perceptron on the test data
y_pred_perceptron = perceptron_model.predict(X_test_scaled)

# Evaluate the accuracy of the Perceptron model
accuracy_perceptron = accuracy_score(y_test, y_pred_perceptron)
print(f"Perceptron Accuracy: {accuracy_perceptron:.4f}")

# Display the confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred_perceptron))

# Display the classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred_perceptron))
```

```
Perceptron Accuracy: 0.8889

Confusion Matrix:
[[19  0  0]
 [ 1  8  4]
 [ 0  0 13]]

Classification Report:
              precision    recall  f1-score   support

           0       0.95      1.00      0.97        19
           1       1.00      0.62      0.76        13
           2       0.76      1.00      0.87        13

    accuracy                           0.89        45
   macro avg       0.90      0.87      0.87        45
weighted avg       0.91      0.89      0.88        45
```

Expected output (with Iris dataset and the proposed parameters):

```
Perceptron Accuracy: 0.8889

Confusion Matrix:
[[19  0  0]
 [ 1  8  4]
 [ 0  0 13]]

Classification Report:
              precision    recall  f1-score   support

           0       0.95      1.00      0.97        19
           1       1.00      0.62      0.76        13
           2       0.76      1.00      0.87        13

    accuracy                           0.89        45
   macro avg       0.90      0.87      0.87        45
weighted avg       0.91      0.89      0.88        45
```

Thinking about it (1)

Given your growing expertise with Perceptron, **answer this questions:**

- What is the shape of the resulting model? Why? Analyze it for both Iris and Breast cancer datasets.

  The shape of the training model for the Iris dataset is (3, 4), while the shape of the Breast cancer one is (1, 30). This is because it represents the trained coeficients with shape n_classes x n_features (notice that in the Breast cancer one there is only 1 row, as it is a binary decision problem, so 2 rows would be redundant). On the other hand, the shape of the predicted model is (45,) for Iris and (171,) for Breast cancer, which are the number of samples used for the test/prediction phase.

- What do you think about the Perceptron accuracy when working with the Iris dataset? How could it be improved?

It is pretty good, but could be better. This could be due to the Perceptron being limited to linearly separable problems, which is not the case with Iris, as only one class is linearly separable from the other two, like it was said in the beginning. This could be improved by using a Multilayer Perceptron instead, which can handle non-linear problems.

- In the confusion matrix, which is the number of Versicolor flowers classified as Virginica?

  The confusion matrix represents the true label along its rows and the predicted label along its columns, so for the Versicolor flowers classified as Virginica we need to look for the second row, third column cell and we see that 4 were classified this way.

- If the model had achieved 100% accuracy, what would the confusion matrix look like?

  It would be a symmetrical matrix with values only in the diagonal and the rest being null. For the Iris matrix, it would have been: [[19 0 0] [0 13 0] [0 0 13]].

- Take a look at the classification report. What precision, recall, f1-score and support mean? And the average values?

  The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The recall is intuitively the ability of the classifier to find all the positive samples. The F1 score can be interpreted as a harmonic mean of the precision and recall. These scores reach their best value at 1 and worst value at 0. The support is the number of true values of each label. The macro average is the unweighted mean per label and the weighted average is the support-weighted mean per label.

Source used for the last question

Limitations

The Perceptron is limited because it can only learn linearly separable problems. If the data is not linearly separable (as is often the case with complex datasets), the Perceptron will struggle to find an accurate solution. This limitation leads to the need for Multilayer Perceptrons (MLPs), which can model non-linear relationships by adding more neurons and layers.

7.4.3 Multilayer Perceptron (MLP)

A **Multilayer Perceptron (MLP)** is an extension of the Perceptron, capable of solving more complex, non-linear problems by introducing multiple layers of neurons and non-linear activation functions. Unlike a single-layer Perceptron, which can only handle linearly separable data, MLPs can approximate any continuous function, thanks to the introduction of **hidden layers**.

Architecture of an MLP

An MLP consists of:

1. **Input layer**: Receives the input features.
2. **Hidden layer(s)**: One or more layers where neurons apply non-linear transformations.
3. **Output layer**: Provides the final prediction, typically using an activation function suited to the task (e.g., softmax for classification).

The key difference between an MLP and a simple Perceptron is the presence of one or more hidden layers, which allow the network to model more complex relationships.

Mathematical Representation

Let's consider a layer of neurons in a hidden or output layer. The input to the MLP is the feature vector $\hat{\mathbf{x}} \in \mathbb{R}^n$, where n is the number of features plus one. Suppose we have a hidden layer with m neurons, and $\mathbf{w}_j^{[l]} = [w_{j1}^{[l]}, w_{j2}^{[l]}, \ldots, w_{jn}^{[l]}, w_{j(n+1)}^{[l]}]^T$ represents the weight vector of the j-th neuron in layer l. The pre-activation value $z_j^{[l]}$ for the j-th neuron in layer l is:

$$z_j^{[l]} = \sum_{i=1}^{n+1} w_{ji}^{[l]} \tilde{a}_i^{[l-1]} = \mathbf{w}_j^{[l]T} \tilde{\mathbf{a}}^{[l-1]}$$

Where:

- $\mathbf{a}^{[l-1]}$ is the output of the previous layer (or the input features for the first hidden layer), plus a 1.
- n is the number of inputs to the layer (either the number of features in the input layer or the number of neurons in the previous hidden layer).

The activation function $g(z)$ is applied to the pre-activation value $z_j^{[l]}$ to produce the output (or activation) of the neuron:

$$a_j^{[l]} = g(z_j^{[l]}) = g(\mathbf{w}_j^{[l]T} \tilde{\mathbf{a}}^{[l-1]})$$

For an entire layer of m neurons, we can represent this compactly using matrix notation:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \tilde{\mathbf{a}}^{[l-1]}$$

Where $\mathbf{W}^{[l]}$ is an $m \times (n+1)$ weight matrix for layer l, where each row corresponds to the weight vector of a neuron in the layer.

Finally, the activation function is applied element-wise to $\mathbf{z}^{[l]}$ to obtain the activations for all neurons in the layer:

$$\mathbf{a}^{[l]} = g(\mathbf{z}^{[l]})$$

Activation Functions

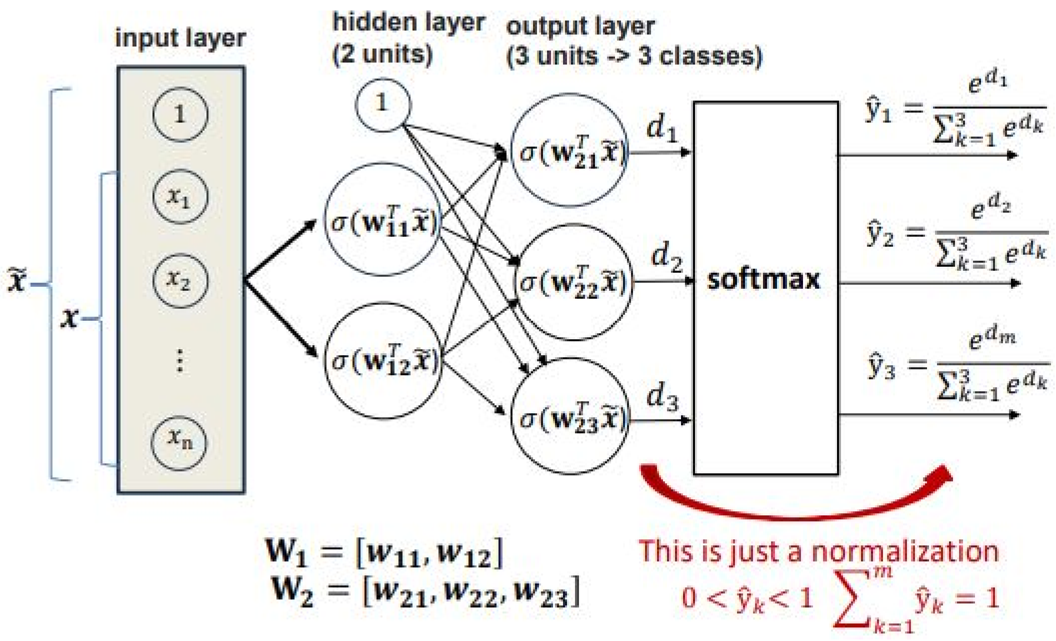
Activation functions introduce non-linearity, allowing the MLP to model more complex patterns. Common activation functions include:

- **Sigmoid**: $g(z) = \frac{1}{1+e^{-z}}$
- **ReLU (Rectified Linear Unit)**: $g(z) = \max(0, z)$
- **Tanh**: $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

For classification tasks, the output layer often uses the **softmax** function to convert raw outputs into probabilities:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Te following figure shows an example of an MLP that classifies among 3 classes with one hidden layer with two neurons (notice that notation slighlty differs since here we have represented the layer of a neuron as a superscript):


input layer
hidden layer
(2 units)
output layer
(3 units -> 3 classes)
$\widetilde{x}$
x
1
x_1
x_2
$\vdots$
x_n
1
$\sigma(\mathbf{w}_{11}^T\widetilde{\mathbf{x}})$
$\sigma(\mathbf{w}_{12}^T\widetilde{\mathbf{x}})$
$\sigma(\mathbf{w}_{21}^T\widetilde{\mathbf{x}})$
$\sigma(\mathbf{w}_{22}^T\widetilde{\mathbf{x}})$
$\sigma(\mathbf{w}_{23}^T\widetilde{\mathbf{x}})$
d_1
d_2
d_3
softmax
$\hat{y}_1 = \frac{e^{d_1}}{\sum_{k=1}^3 e^{d_k}}$
$\hat{y}_2 = \frac{e^{d_2}}{\sum_{k=1}^3 e^{d_k}}$
$\hat{y}_3 = \frac{e^{d_m}}{\sum_{k=1}^3 e^{d_k}}$
$\mathbf{W}_1 = [\mathbf{w}_{11}, \mathbf{w}_{12}]$
$\mathbf{W}_2 = [\mathbf{w}_{21}, \mathbf{w}_{22}, \mathbf{w}_{23}]$
This is just a normalization
$0 < \hat{y}_k < 1 \quad \sum_{k=1}^{m} \hat{y}_k = 1$

Backpropagation and Learning

MLPs are trained using **backpropagation**, which computes the gradient of the loss function with respect to the model's weights and biases. The gradients are used to update the parameters using **gradient descent**.

Loss Function

For classification tasks, the loss function is often the **cross-entropy loss**. Given a training sample i:

$$L_i(\mathbf{W}) = -y_i \ln(\hat{y}_i(\mathbf{W}))$$

Where:

- y_i is the true label (one-hot encoded, e.g, $y_i = [0, \ldots, 1, \ldots, 0]$).

- $\hat{y}_i$ is the vector of predicted probabilitie for each class.

Cross-entropy measures the difference between the true labels and the predicted probabilities in classification tasks, penalizing confident but incorrect predictions more heavily while rewarding confident and correct ones.

This way, the goal is to minimize the average of the loss across all the N samples in the dataset, which is called the **cost function**:

$$\hat{W} = \mathrm{argmin}_{\mathbf{W}} L(\mathbf{W}) = \mathrm{argmin}_{\mathbf{W}} \frac{1}{N} \sum_{j=1}^{N} L_j$$

Weight Update Rule

The weights are updated as follows:

$$w^{[l]} \leftarrow w^{[l]} - \eta \frac{\partial L}{\partial w^{[l]}}$$

Where:

- η is the learning rate.
- $\frac{\partial L}{\partial w^{[l]}}$ is the gradient of the loss with respect to the weights at layer l.

The backpropagation algorithm ensures that the gradients are propagated backward from the output layer to the input layer, updating the parameters at each step.

ASSIGNMENT 3: MLP training and model analysis

Scikit-learn provides the class `MLPClassifier` that implements a multi-layer perceptron (MLP) algorithm that trains using gradient descent and Backpropagation with Corss-Entropy loss function.

This class accepts many parameters in its constructor. You can explore them here, but some relevant are:

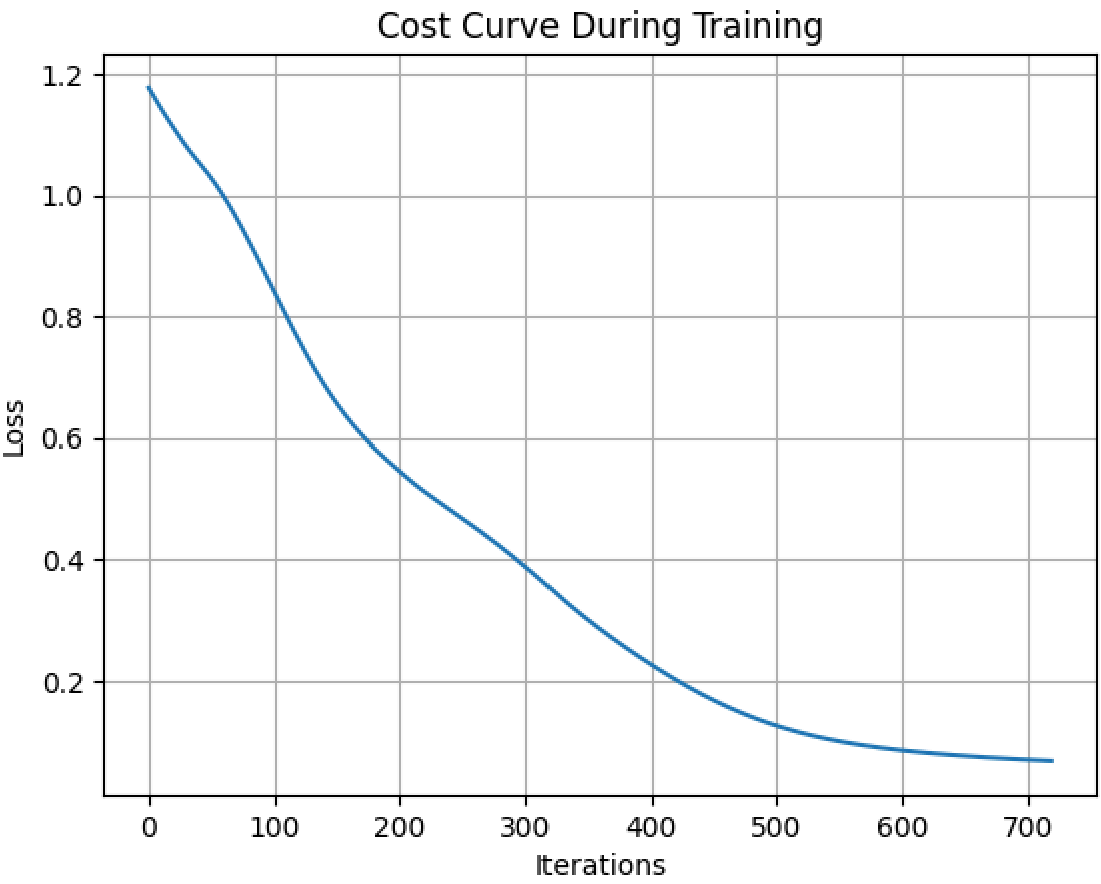
- `hidden_layer_sizes (array-like)` : represents the number of neurons in the ith **hidden layer**. For example if you want to design a network with 5 neurons in the first hidden layer and 4 in the second one, you must introduce `(5,4)` . Hint: The number of neurons in the **output layer** can not be set since it would depend on the number of output categories.
- `activation {‘identity’, ‘logistic’, ‘tanh’, ‘relu’}, default=’relu’` : activation function for the hidden layers.
- `max_iter (int, default=200)` : maximum number of iterations. Sometimes it's a good idea to increase it.

Complete the following code to create the model object (`mlp`) and train it. To check if your results are right, design a networ with 10 neurons in the first hidden layer, and 5 in the second one, use ReLu as the activation function, a maximum number of iterations of

1000, and a random state of 42. After checking the results you can try other parameters. It is also reported information about the evolution of the cost function during training.

In [7]:
```
# Initialize and train the MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(10,5),
                    activation='relu',
                    max_iter=1000,
                    random_state=42)

mlp.fit(X_train_scaled, y_train)

print(f"Final loss: {mlp.loss_}")  # Final loss value after training
print(f"Number of iterations: {mlp.n_iter_}")  # Number of iterations the model
print(f"Best loss: {mlp.best_loss_}")  # Best loss (when early stopping is used)

# Plot the loss curve
plt.plot(mlp.loss_curve_)
plt.title('Cost Curve During Training')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.grid()
plt.show()
```

```
Final loss: 0.06797965925841047
Number of iterations: 720
Best loss: 0.06797965925841047
```



Expected output (with Iris dataset and the proposed parameters):

```
Final loss: 0.06797965925841051
Number of iterations: 720
Best loss: 0.06797965925841051
```

The model coeficients (**weights**) are stored in the `coefs_` attribute. It provides valuable insight about the network structure, so let's take a closer look at it. Another key attribute is `intercepts_`, which is where the library stores the weights associated with the bias terms (the constant 1s added to the input vector and the activations of each layer).

In [8]:

```
print('Number of hidden layers:', len(mlp.coefs_)-1) # We have to substract the

print('Size of Weight matrices at each layer')
for coef in mlp.coefs_:
    print (coef.shape)

print('Weights associated to bias terms (intercepts):')
for intercept in mlp.intercepts_:
    print(intercept.shape)
```

```
Number of hidden layers: 2
Size of Weight matrices at each layer
(4, 10)
(10, 5)
(5, 3)
Weights associated to bias terms (intercepts):
(10,)
(5,)
(3,)
```

Expected output (with Iris dataset and the proposed parameters):

```
Number of hidden layers: 2
Size of weight matrices at each layer
(4, 10)
(10, 5)
(5, 3)
Weights associated to bias terms (intercepts):
(10,)
(5,)
(3,)
```

ASSIGNMENT 4: Analyzing MLP performance

Let's complete the following code to analyze the model performance. For that:

- Call the `predict()` function with the right data.
- Show the first 5 predicted labels along with their corresponding true labels.
- Prints its accuracy, with `accuracy_score()` (docs),
- The confusion matrix, with `confusion_matrix()` (docs),
- And the classification report with `classification_report()` (docs).

In [9]:

```
# Make predictions on the test set
y_pred = mlp.predict(X_test_scaled)

# Display first 5 predictions
print("First 5 predicted labels:", y_pred[:5])
print("Corresponding true labels:", y_test[:5])
```

```
# Assignment: Calculate and print the accuracy score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy on the test set: {accuracy:.4f}")

# Evaluate the model performance using classification metrics
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
First 5 predicted labels: [1 0 2 1 1]
Corresponding true labels: [1 0 2 1 1]
Accuracy on the test set: 1.0000
Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Expected output (with Iris dataset and the proposed parameters):

```
First 5 predicted labels: [1 0 2 1 1]
Corresponding true labels: [1 0 2 1 1]
Accuracy on the test set: 1.0000
Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Thinking about it (2)

Given your growing expertise with MLP, **answer this questions:**

- How many layers has the rained MLP? Of which type?

  It technically has 4 layers: the input layer, 2 hidden layers and the output layer, but there are only 3 weight matrices, those corresponding to the 2 hidden layers and the output layer.

- What is the size of each weight matrix, and how does it relate to the input features, hidden layers, and output classes?

  The sizes of the weighted matrices are: (4, 10), (10, 5) and (5, 3). The first size represents the 4 features of the input layer and the 10 neurons of the first hidden layer. The second size represents the 10 outputs of the neurons in the first hidden layer and the 5 neurons of the second hidden layer. Lastly, the third size represents the 5 outputs of the neurons in the second hidden layer and the 3 classes the output has to classify.

- Given that the output layer has more than a neuron, how is the final MLP output computed?

  After applying the non-linear activation function, since it is a classification problem, the softmax function is applied so that every output is converted to a probability (a value between 0 and 1). That is, normalising the raw outputs.

A final note on softmax

Since the trained model has the `softmax` activation function in the output layer, so you can retrieve probabilities about the classification using the `predict_proba()` function.

The following code illustrate this:

In [10]:

```
print(mlp.out_activation_)

y_pred = mlp.predict_proba([X_test_scaled[0,:]])

formatted_probas = [f"{prob:.2%}" for prob in y_pred[0]]
print(f"Predicted probabilities for the first sample (as percentages): {formatte
```

```
softmax
Predicted probabilities for the first sample (as percentages): ['0.09%', '97.8
2%', '2.09%']
```

OPTIONAL

You can try the Perceptron and MLP performance with any other dataset of your choice. There are may computer vision-related datasets on the internet!

END OF OPTIONAL PART

OPTIONAL

scikit-learn has many different classification techniques. You can try other models and see how they perform.

END OF OPTIONAL PART

Summary

In this notebook, we used a Perceptron and a Multilayer Perceptron to classify samples from different datasets, like the Iris flowers species and the Breast Cancer detection. We explored data preprocessing steps, trained those models, and evaluated them.

This is just the beginning! The neural networks journey is vast...