The dataset is organized in two directories, according to the structure:

```
casting_data
├──test
|   ├──def_front
|   └──ok_front
└──train
    ├──def_front
    └──ok_front
```

This structure is convenient because, at the time of reading by keras, the classes def_front and ok_front are recognized according to the directories names.

```
technocast_PATH = '/kaggle/input/real-life-industrial-dataset-of-casting-product/casting_data/casting_data/'

technocast_train_path = technocast_PATH + 'train/'
technocast_test_path = technocast_PATH + 'test/'
```

Using matplotlib to take a look at two random pieces (normal and defective):

```
dir1 = technocast_train_path+'/ok_front/'
dir2 = technocast_train_path+'/def_front/'

img1 = plt.imread(dir1+random.choice(os.listdir(dir1)))
img2 = plt.imread(dir2+random.choice(os.listdir(dir2)))

fig, ax = plt.subplots(1,2)

ax[0].imshow(img1)
ax[0].axis('off')
ax[0].set_title('ok')
```

```
ax[1].imshow(img2)
ax[1].axis('off');
ax[1].set_title('defective');
```

Our models will need to distinguish these two types of images.

Generating data with Keras

Image files must be converted to tensors to be fed into neural networks (we will learn more about tensors below).

We will use the class ImageDataGenerator (available in the module keras.preprocessing.image) to generate the input tensors from the files availables in the technocast_train_path andtechnocast_test_path folders.

This generation will take place in real time during training: at each iteration, a minibatch of tensors will be provided to the model so that a learning step can be carried out.

In the function below, we instantiate an object named datagen from theImageDataGenerator class, specifying that the pixels will be normalized to the 0-1 range (by dividing by 255) and 10% of the data will be reserved for validation. After that, we use the flow_from_directory method to effectively create the objects that will generate the training and validation minibatches.

```
def make_generators():

    datagen = keras.preprocessing.image.ImageDataGenerator(rescale = 1/255,
                                                           validation_split = 0.1)

    train_generator = datagen.flow_from_directory(directory = technocast_train_path,
                                                  batch_size = 32,
                                                  target_size = (300, 300),
                                                  color_mode = "grayscale",
                                                  class_mode = "binary",
                                                  classes = {"ok_front": 0, "def_front": 1},
                                                  shuffle = True,
                                        #seed = 0,
                                        subset = "training")

    validation_generator = datagen.flow_from_directory(directory = technocast_train_path,
                                          batch_size = 32,
                                          target_size = (300, 300),
                                          color_mode = "grayscale",
                                          class_mode = "binary",
                                          classes = {"ok_front": 0, "def_front": 1},
                                          shuffle = True,
                                          #seed = 0,
                                          subset = "validation")

    return train_generator, validation_generator
```

The arguments of the flow_from_directory method specify that minibatches will be generated containing 32 grayscale images (that is, with only 1 color channel) of shape 300x300. The classification problem is binary; the normal (ok_front) and defective (def_front) classes were assigned labels 0 and 1, respectively. The data will be shuffled and the seed of randomness has been specified as 0.

At this point, it is convenient to take a look at an example of a minibatch. For this, we will use the function visualizeImageBatch:

```
def visualizeImageBatch(datagen, title):

    '''
    Adapted from:
    https://www.kaggle.com/tomythoven/casting-inspection-with-data-augmentation-cnn
    '''

    mapping_class = {0: "0 (ok)", 1: "1 (defective)"}

    images, labels = next(iter(datagen))
    images = images.reshape(32, *(300,300))

    fig, axes = plt.subplots(4, 8, figsize=(13,6.5))

    for ax, img, label in zip(axes.flat, images, labels):
        ax.imshow(img, cmap = "gray")
        ax.axis("off")
        ax.set_title(mapping_class[label], size = 12)

    fig.tight_layout()
    fig.suptitle(title, size = 16, y = 1.05)
```

Inspecting a manibatch:

```
train_generator, validation_generator = make_generators()

visualizeImageBatch(train_generator, 'Training manibatch example')
```

```
Found 5971 images belonging to 2 classes.
Found 662 images belonging to 2 classes.
```

Now let's create the test image generator, defined a bit differently:

```
test_datagen = keras.preprocessing.image.ImageDataGenerator(rescale = 1/255)

n_test = sum([len(files) for r, d, files in os.walk(technocast_test_path)])

test_generator = test_datagen.flow_from_directory(directory = technocast_test_path,
                                                  batch_size = n_test,
                                                  target_size = (300, 300),
                                                  color_mode = "grayscale",
                                                  class_mode = "binary",
                                                  classes = {"ok_front": 0, "def_front": 1},
                                             shuffle = False)
```

```
Found 715 images belonging to 2 classes.
```

We specified the minibatch size as the total size of the test set, since the test prediction will be made with all the data at once, after training ends.

It is useful to check the proportions of classes in the different sets:

```
fig, ax = plt.subplots(1,3,figsize=(10,3))

sns.countplot(train_generator.classes,ax=ax[0])
sns.countplot(validation_generator.classes,ax=ax[1])
sns.countplot(test_generator.classes,ax=ax[2])

ax[0].set_title('Training')
ax[1].set_title('Validation')
ax[2].set_title('Test')

fig.suptitle('Proportion of classes (normal/defectives)')
fig.tight_layout(rect=[0, 0.03, 1, 0.92]);
```

The proportions seem consistent between sets. We could try to correct the prevalence of defective part images over normal ones, but as the difference is not so great, it will not be done.

Ok!! We already have the objects that will generate our training, validation and test data. Now let's proceed to the modeling phase.

Convolutional networks

Convolutional networks are sparsely connected networks, where each neuron does not connect to all neurons in the previous layer. It is the opposite of densely connected networks, such as MLP.

Densely connected networks can only learn global patterns, that is, patterns that involve all points present in a given sample or observation. Convolutional networks, as we will see, are able to learn local patterns and recognize these patterns in any position of some other sample. In the case of images, these local patterns may correspond to borders, curves, etc.

Throughout the network structure, as we advance into deeper layers, simple local patterns are aggregated and become more and more complex. It is a learning structure analogous to that observed in the visual cortex, the brain portion responsible for the processing of visual information. That is why convolutional networks are so used to process images.

Mathematically, the network learns these local patterns through the convolution operation, explained below.