SAPIENZA
UNIVERSITÀ DI ROMA

Master of Science in Engineering in Computer Science

Machine Learning

2020/2021

Valerio Valente | 1894954

HOMEWORK 2

Latina, Italy

Abstract

This report describes the building process and development phases adopted to accomplish the assigned homework. In order to achieve the objective, we have studied the *Deep Learning* methods: *Neural Networks* (*NN*), and especially those called *Convolutional Neural Networks* (*CNN*). The goal was trying to classify a dataset containing 8 different classes of objects typically available in a home environment.

Introduction

First of all, we start this report by describing what Deep Learning is.
Deep Learning is a specific subfield of Machine Learning: a new take on learning representations from data that puts an emphasis on learning successive *layers* of increasingly meaningful representations. The 'deep' in Deep Learning isn't a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations. How many layers contribute to a model of the data is called the *depth* of the model.
In deep learning, these layered representations are (almost always) learned via models called Neural Networks, structured in literal layers stacked on top of each other. The term Neural Network is a reference to neurobiology, in particular to the idea of the anatomical structure of a neuron.
What do the representations learned by a deep-learning algorithm look like? Let's examine how a network several layers deep (see figure 1) transforms an image of a digit in order to recognize what digit it is.

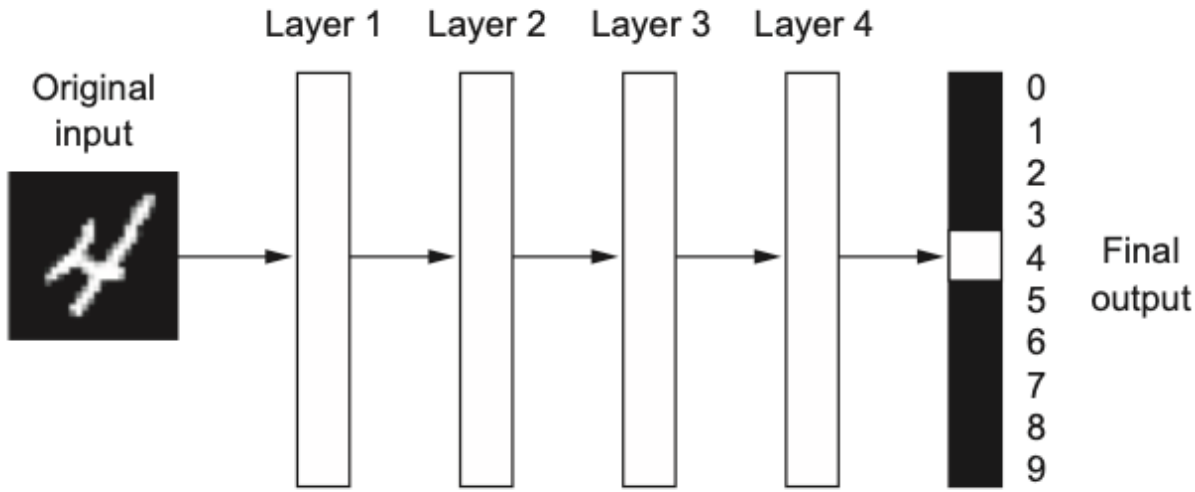


Figure 1

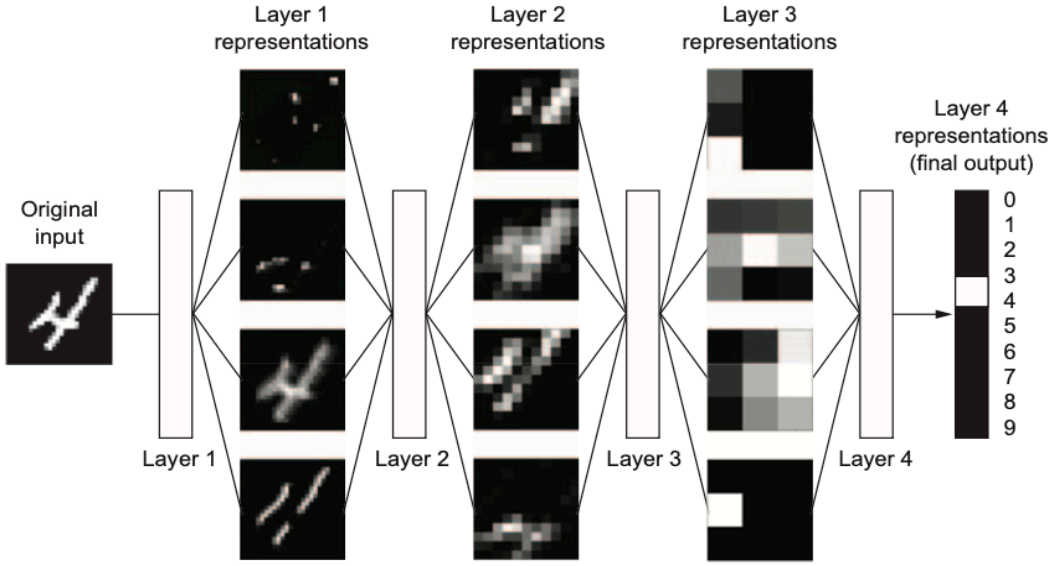


Figure 2

As you can see in figure 2, the network transforms the digit image into representations that are increasingly different from the original image and increasingly informative about the final result. You can think of a deep network as a multistage information-distillation operation, where information goes through successive filters and comes out increasingly *purified*.

The specification of what a layer does to its input data is stored in the layer's *weights*, which in essence are a bunch of numbers. In technical terms, we'd say that the transformation implemented by a layer is parameterized by its weights. (Weights are also sometimes called the *parameters* of a layer.) In this context, learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets.

To control the output of a neural network, you need to be able to measure how far this output is from what you expected. This is the job of the *loss function* of the network, also called the *objective function*. The loss function takes the predictions of the network and the true target (what you wanted the network to output) and computes a distance score, capturing how well the network has done on this specific example.

The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example. This adjustment is the job of the *optimizer*, which implements what's called the *Backpropagation algorithm*: the central algorithm in deep learning (figure 3 shows the whole model).

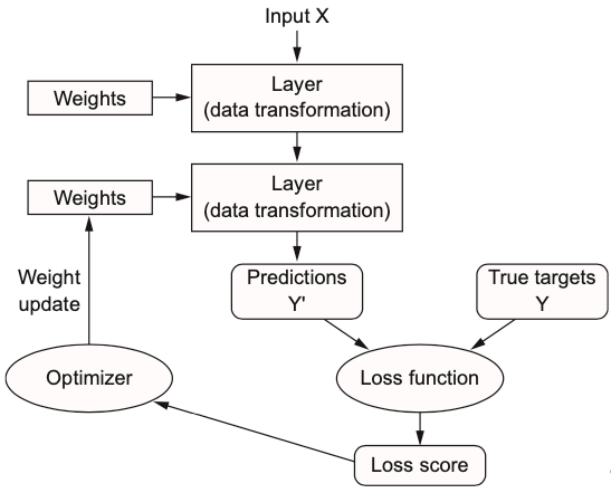


Figure 3

Convolutional neural networks, also known as *convnets*, are a type of deep-learning model almost universally used in computer vision applications.
The fundamental element of a CNN is a *convolution layer* which is specialized in learn local patterns through the *convolution* operation.
Convolutions operate over 3D tensors, called *feature maps*, with two spatial axes (*height* and *width*) as well as a depth axis (also called the *channels axis*). For an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an *output feature map*. This output feature map is still a 3D tensor: it has a width and a height. Its depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, they stand for *filters*. Filters encode specific aspects of the input data: at a high level, a single filter could encode the concept "presence of a face in the input," for instance.
Convolutions are defined by two key parameters:

- Size of the patches extracted from the inputs, typically 3×3 or 5×5;
- Depth of the output feature map: the number of filters computed by the convolution.

Design

We first started by splitting the provided dataset, adopting a 0.7 for the training set and a 0.3 for the validation set. Plus, we took 4 images for each class of elements from the training set and put them into a third set which will be used for blind test.

In order to have a fully random split we installed the library "Split-Folders" through the *pip* shell command. We operated for the whole time over our mounted Google Drive cloud, especially useful to be navigated without too many issues.

So, we had 3 sets: Training Set, Validation Set and Test Set.

At this point we had to preprocess somehow all the images for both training and validation set, in order to make them "readable" from the CNN.

In fact, data should be formatted into appropriately preprocessed floating- point tensors before being fed into the network. We had the data stored on a drive as JPEG files, so the steps for getting them into the network were roughly as follows:

1. Read the picture files;
2. Decode the *JPEG* content to *RGB* grids of pixels;
3. Convert these into floating-point tensors;
4. Rescale the pixels values (between 0 and 255 to the [0,1] interval).

The *Keras* library has utilities to take care of these steps automatically.

Keras has a module with image-processing helper tools, located at *keras.preprocessing.image*. In particular, it contains the class *ImageDataGenerator*, which lets us quickly set up Python generators that can automatically turn image files on disk into *batches* of preprocessed tensors.

Now let's give a glance to the CNN built for the homework, it is a stack of alternated *Conv2D* (with *relu* activation) and *MaxPooling2D layers*

(see Figure 4). Since we were facing a classification problem with more than 2 labels available, we adopted for the compilation step, the *adam* optimizer. Because we ended the network with a *softmax* unit, we used *categorical-crossentropy* as the loss.

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',\
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
# 8 and softmax, why?
model.add(layers.Dense(8, activation='softmax'))

model.summary()
```

Figure 4

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 7, 7, 128)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_6 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_7 (Dense)              (None, 8)                 4104
=================================================================
Total params: 3,456,712
Trainable params: 3,456,712
Non-trainable params: 0
_________________________________________________________________
```

Figure 5

We trained our model, adopting 10 as number of epochs. This was due to both the sluggish performances of our IDE, and the willingness to check the behavior of our CNN. We got a good accuracy and a decreasing loss function until epoch 5. After that, the model began to behave badly, increasing and decreasing continuously the values of accuracy and loss function. (For comprehension sake we want to specify that, when we are talking about such values above, we refer to those of the validation set).

In figure 6, you can see the whole trend of the values compared to those of the training set. It was clear that we were having some overfitting troubles.

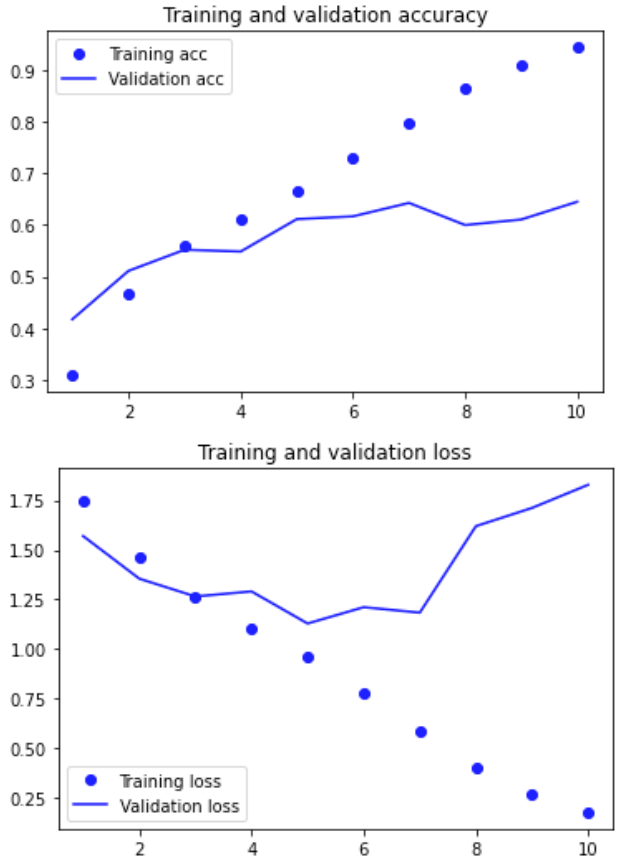


Figure 6

These plots above, are characteristic of overfitting. The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy stalls at 64–65%. The validation loss reaches its minimum, whereas the training loss keeps decreasing linearly until epoch 5, when it starts to grow again.

In order to solve such problem, we decided to adopt one of the many available techniques to reduce overfitting, in particular (because we have not so many data) the *Data Augmentation*.

Data augmentation takes the approach of generating more training data from existing training samples, by augmenting the samples via a number of random

transformations that yield believable-looking images. The goal is that at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data and generalize better.
In Keras, this can be done by configuring a number of random transformations.
In figure 7, we show one of our transformed images.

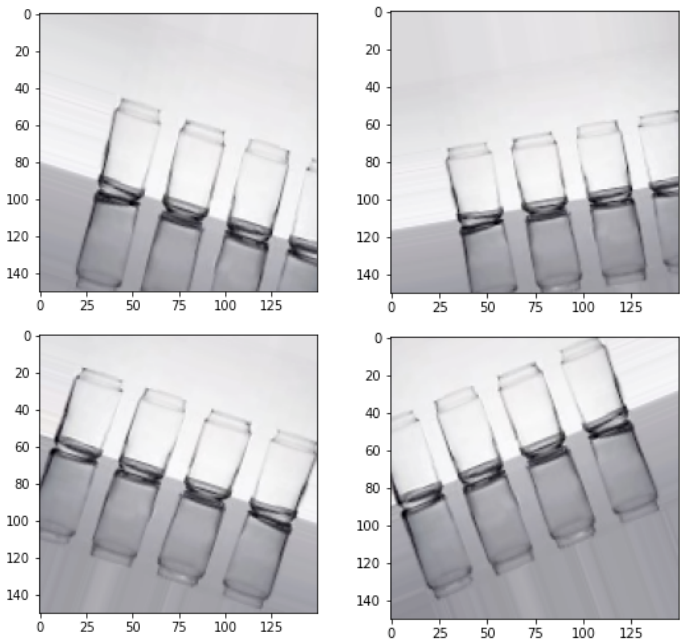


Figure 7

Hence, we trained a new CNN to which we have added a *Dropout* layer to increase the reduction of overfitting. In Figure 8 you can observe the model.

```
from keras import layers
from keras import models

# To fight overfitting, we add a Dropout layer to the model
model2 = models.Sequential()
model2.add(layers.Conv2D(32, (3, 3), activation='relu',\
                         input_shape=(150, 150, 3)))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Conv2D(64, (3, 3), activation='relu'))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Conv2D(128, (3, 3), activation='relu'))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Conv2D(128, (3, 3), activation='relu'))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Flatten())
model2.add(layers.Dropout(0.2))
model2.add(layers.Dense(512, activation='relu'))
# 8 and softmax, why?
model2.add(layers.Dense(8, activation='softmax'))

model2.summary()

model2.compile(optimizer='adam',\
               loss='categorical_crossentropy',\
               metrics=['accuracy'])
```

Figure 8

Notice that the structure is pretty identical to the former one except for the dropout layer. We tested our CNN and came up with a decent result in about 22 epochs. As you can see in figure 9, we got a good accuracy around 68-70% and the loss function decreased down to ≅0.95. The main difference with respect to the previous model can be seen especially in the trend graphics (Figure 9). The values of both training and validation set proceeded jointly to the last epoch.

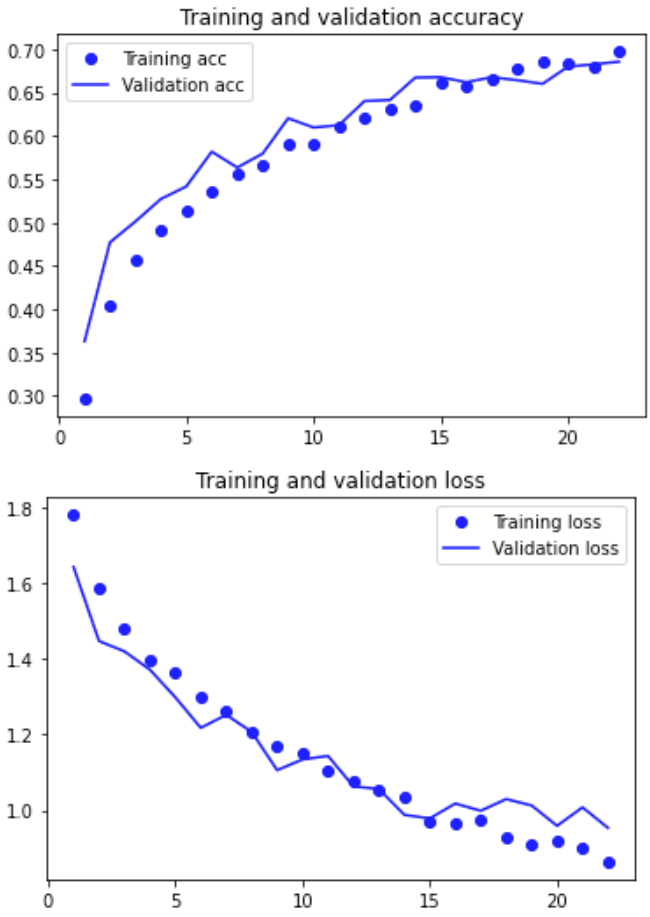
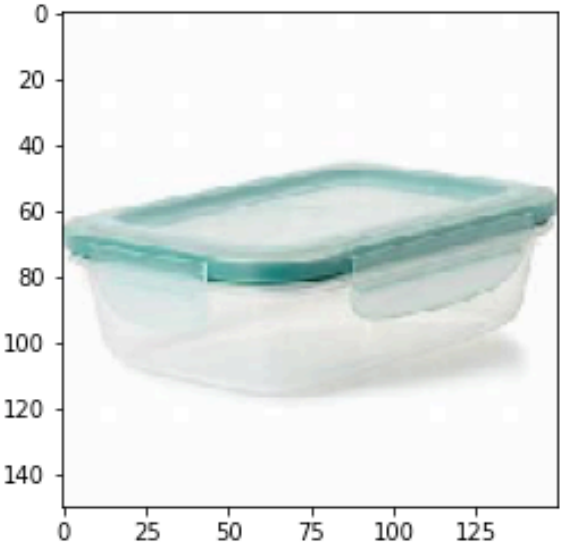
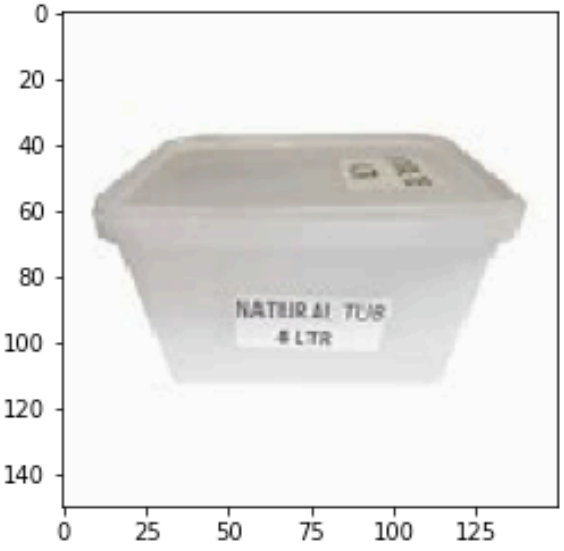


Figure 9

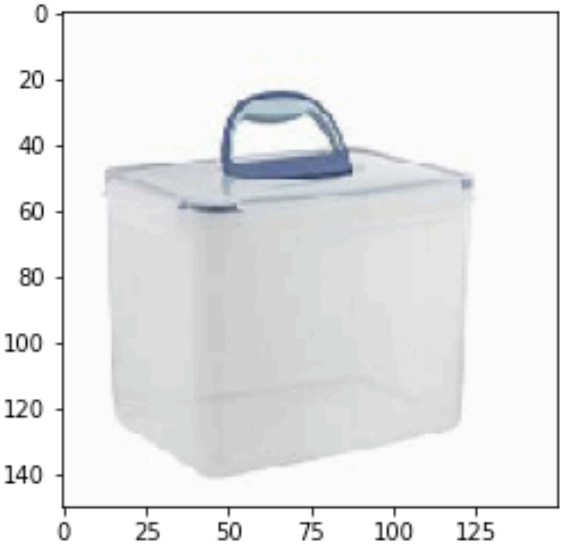
With such a model we were quite satisfied, hence we saved it through the specific function and evaluated the test set with the *predict* function.
What follows is the whole prediction schema, you can see the plotted image and its predicted label below.



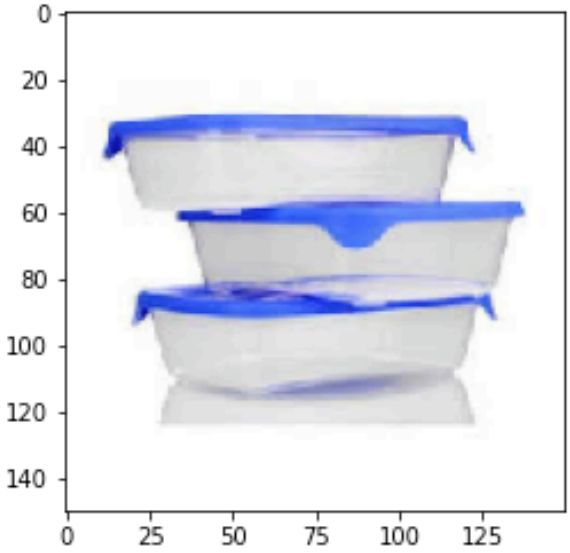
plastic_food_container



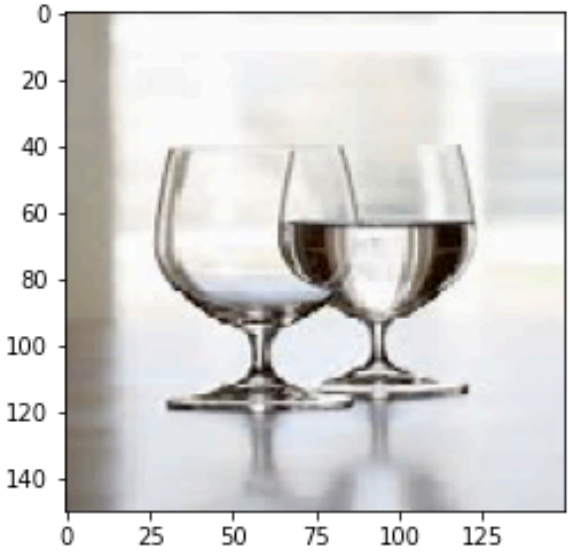
plastic_food_container



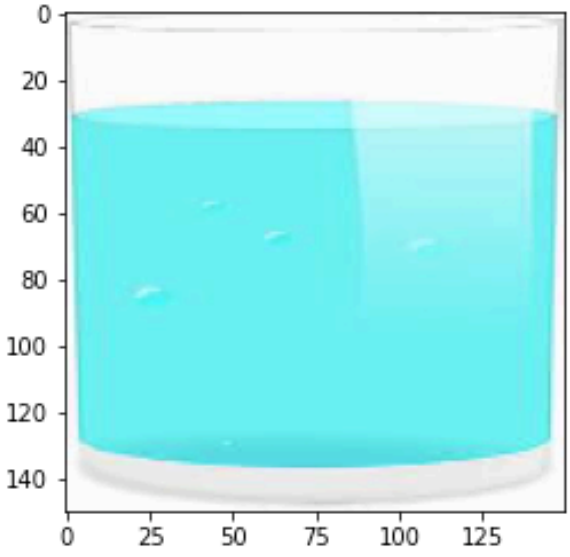
plastic_food_container



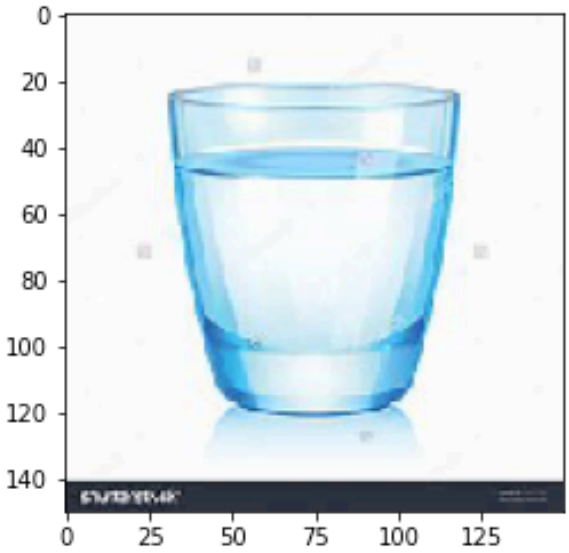
plastic_food_container



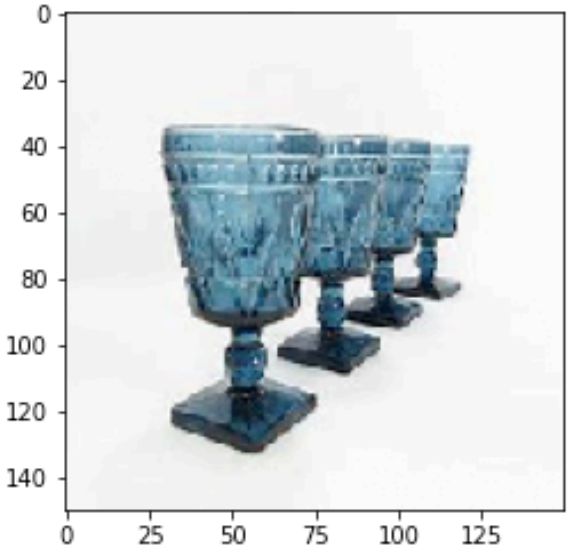
water_glasses



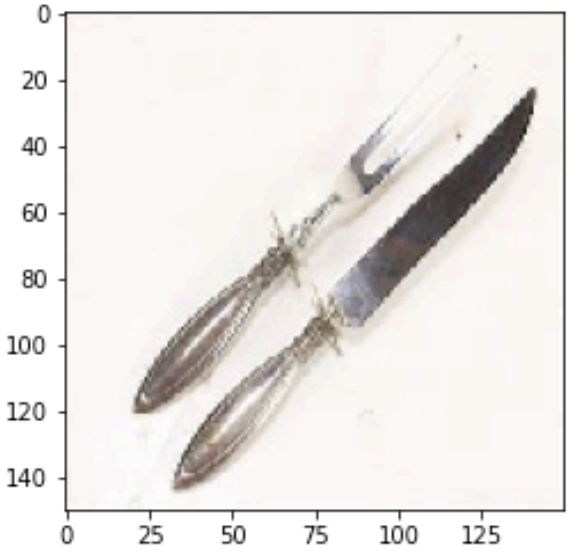
water_glasses



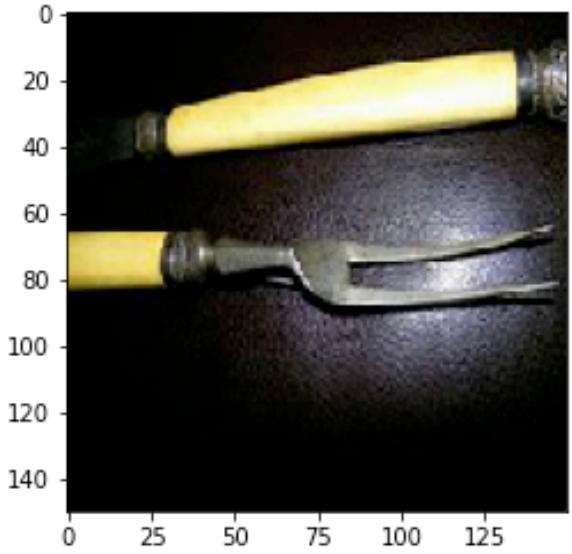
water_glasses



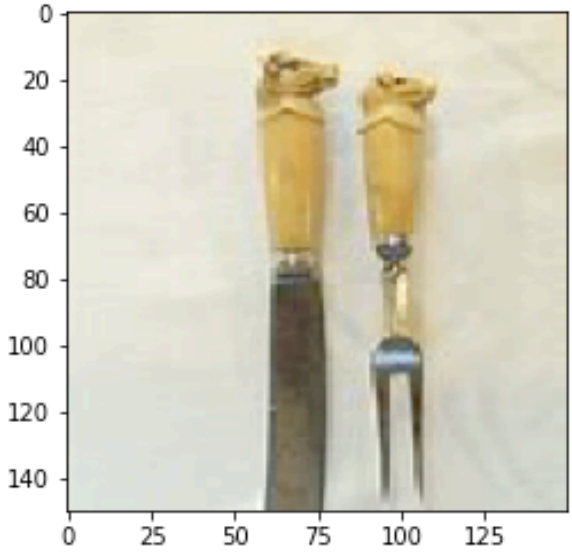
water_glasses



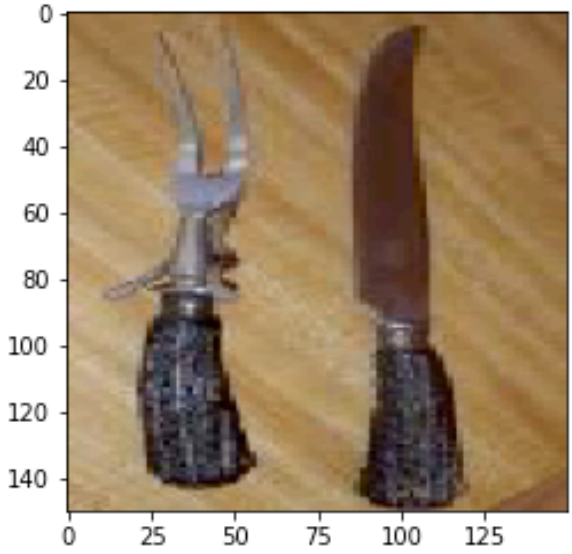
carving_knife_fork



carving_knife_fork

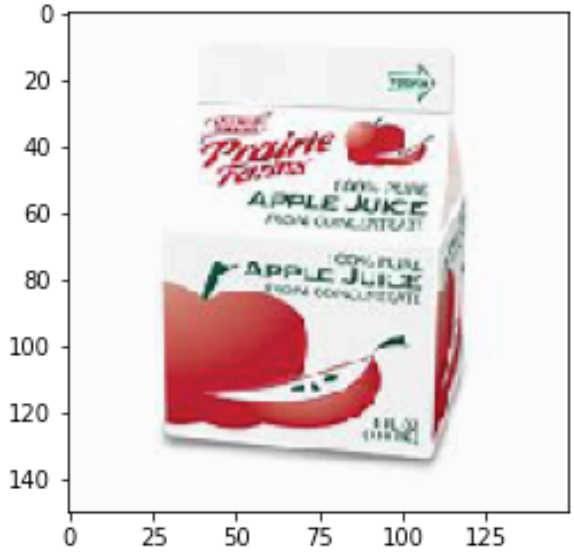


Lollipops

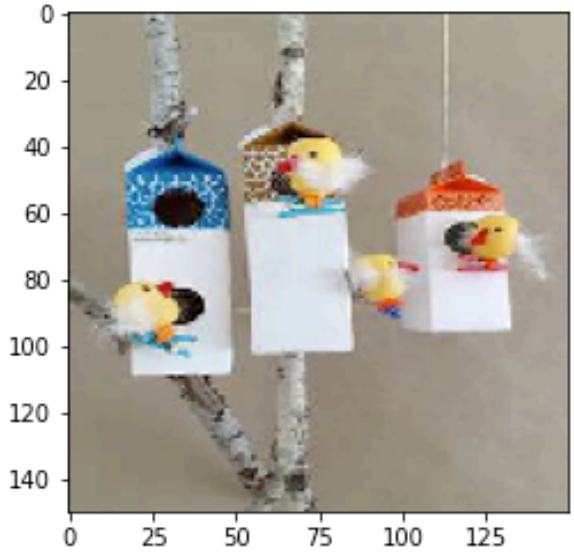


carving_knife_fork

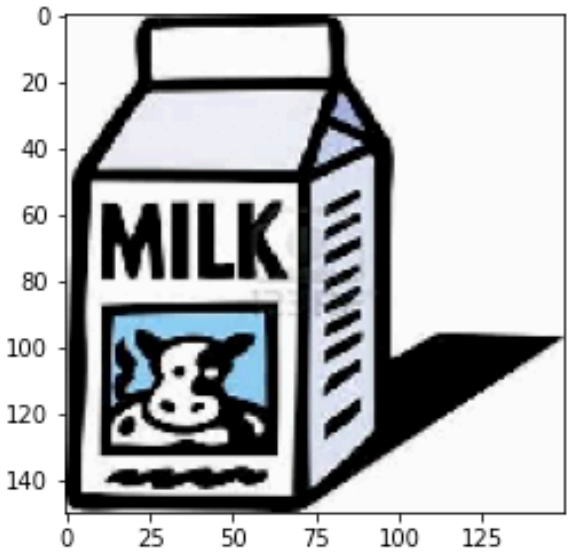
carving_knife_fork



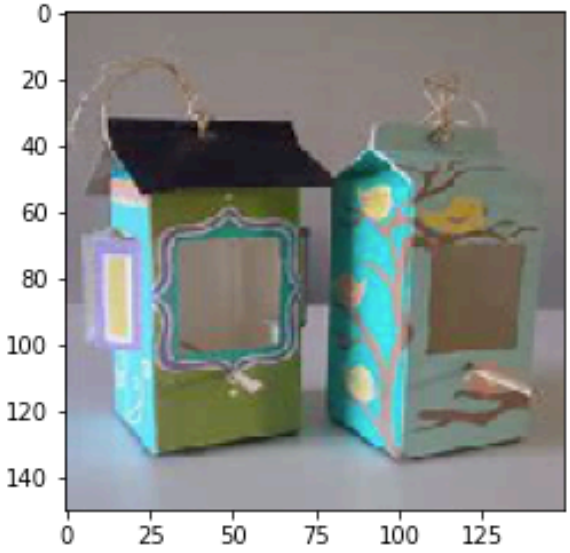
juice_carton



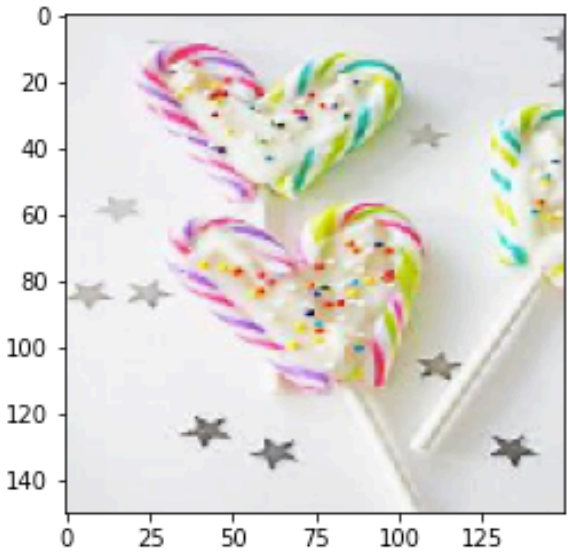
Caddies




```
juice_carton
```

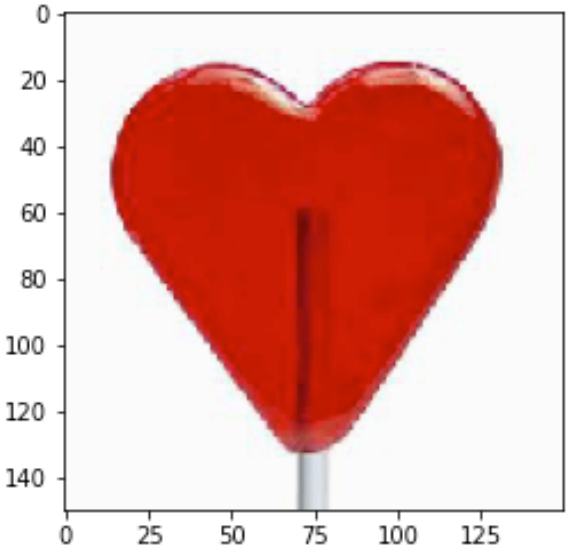


```
juice_carton
```

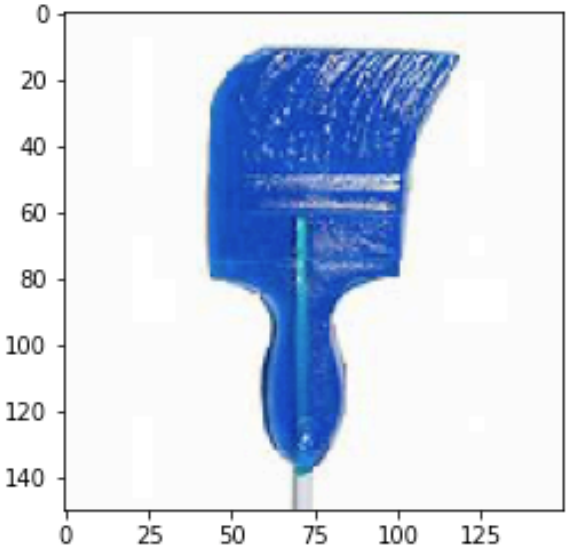


```
Lollipops
```

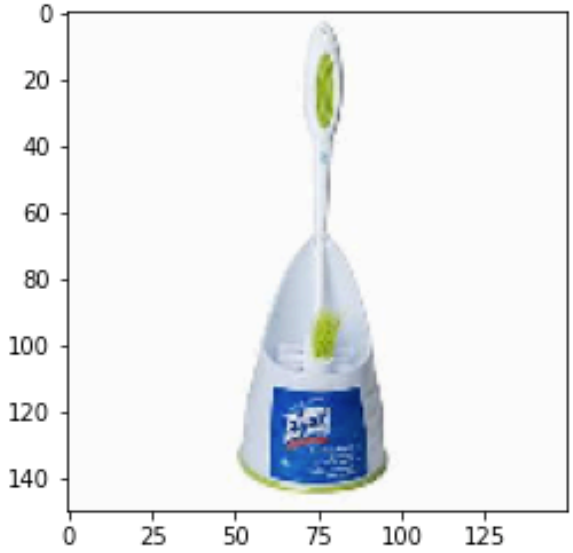
Lollipops



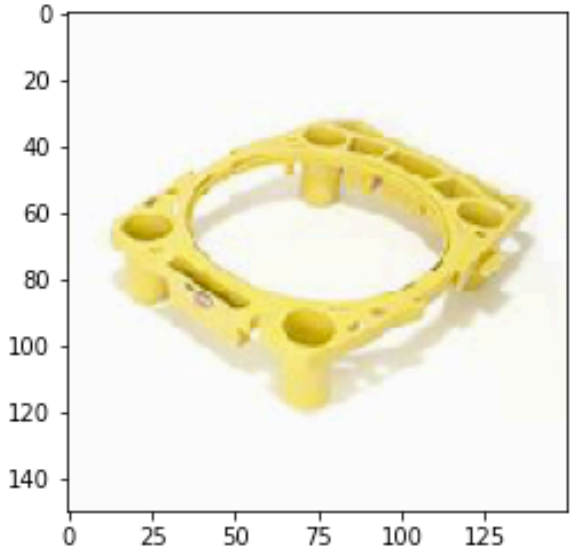
Caddies



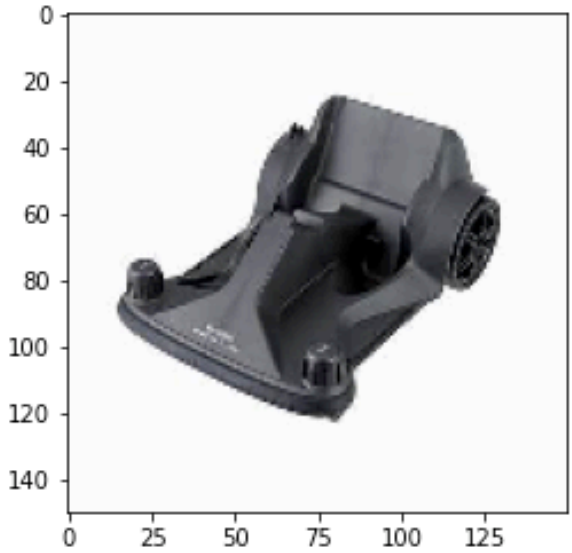
water_glasses



water_glasses



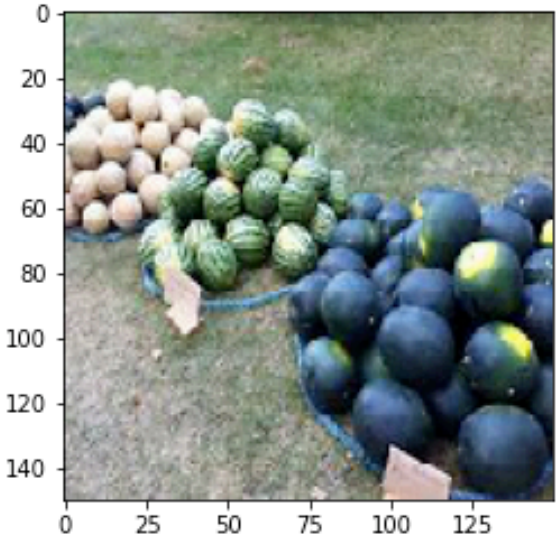
Melons



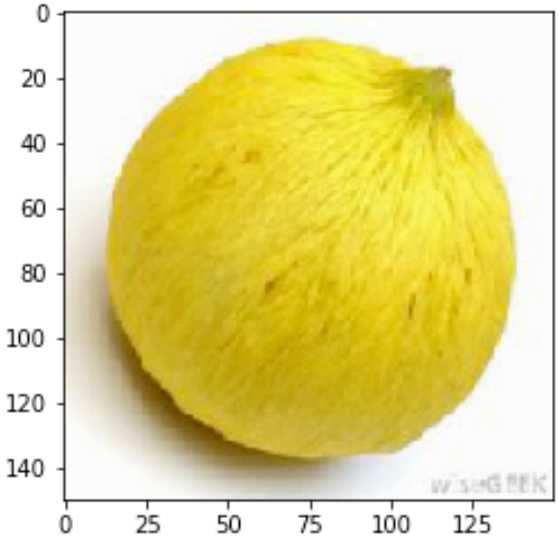
Caddies

Caddies

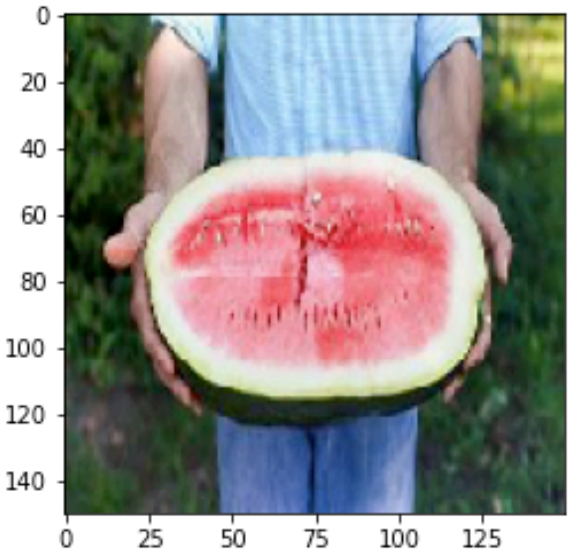
Caddies



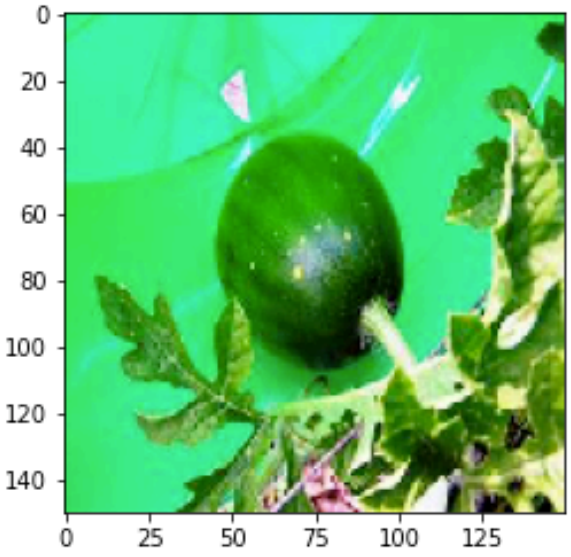
Caddies



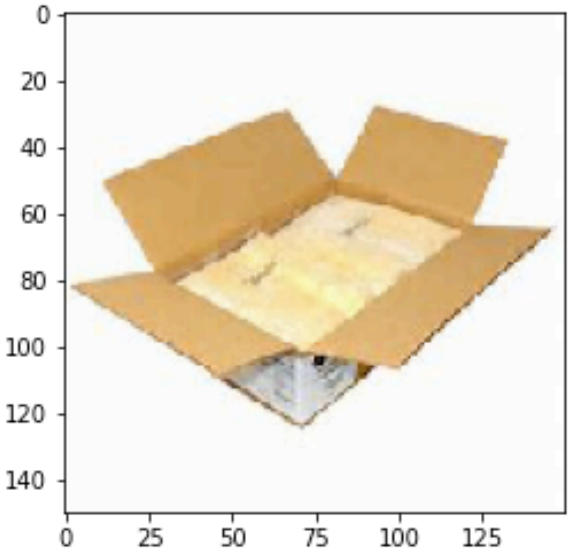
Melons




```
Melons
```

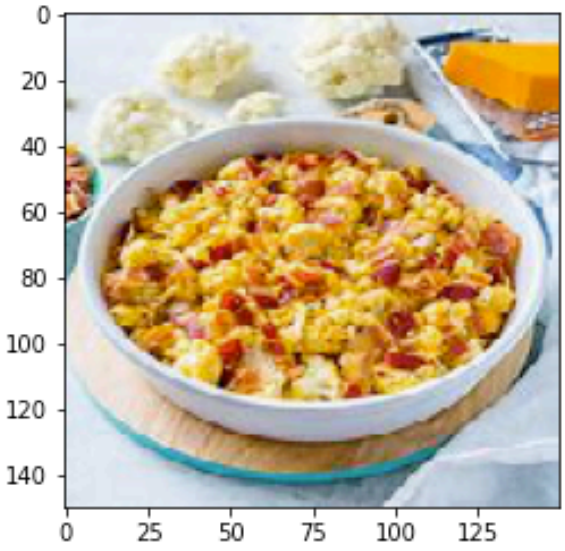



```
Melons
```

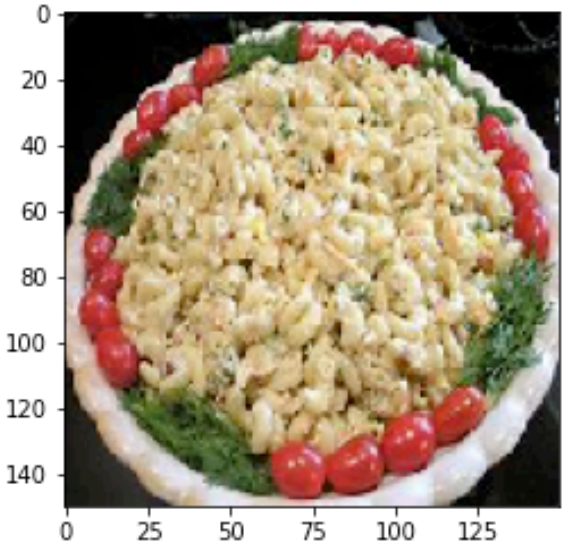



```
Melons
```

Macaroni_&_Cheese_box



Macaroni_&_Cheese_box



Macaroni_&_Cheese_box