Artificial Intelligence - CET4

Neural Networks and Deep Learning

Juan José Rodríguez Aldavero

May 27, 2020

Contents

1 Exploratory data analysis

In this deliverable we will apply Deep Learning techniques to study the CIFAR-10 image dataset. In particular, we will train and apply Convolutional Neural Networks (CNN) to classify images of the CIFAR-10 dataset.

First, we begin the practice by performing a brief exploratory analysis on the data set. To do this, we create a group of functions that allow us to load the dataset and visualize the images it contains. We wanted to make all these functions so that they could be generalized for a different set of images, and for this reason parameters such as the titles have to be specified externally.

A. **Create a code to load the dataset as numpy arrays**
We load the images from the CIFAR10 dataset using a predefined method in the TensorFlow library. This is separated by default into a training set and a test set, with 50000 training images and 10000 test images. Each image has dimensions of $(32 \times 32 \times 3)$ pixels in raster format, where there are three color channels and each pixel can take values between 0 and 255.

```
def load_cifar10():
  import tensorflow as tf
  cifar10 = tf.keras.datasets.cifar10.load_data()
  x_train = cifar10[0][0]
  y_train = cifar10[0][1]
  x_test = cifar10[1][0]
  y_test = cifar10[1][1]
  return x_train, y_train, x_test, y_test
```

B. **Build a 2×5 figure containing the first image of each class.**
We proceed by creating a function that displays the first image of each class in grid format. We observe how a list must be passed to the parameter *titles* to define the titles, and otherwise they appear as "Unknown". The parameter *n_image* defines the number of the image of each class that you want to show.

```
def inspect_images(X, y, n_image=0, titles = None, columns=5):
  import numpy as np
  import matplotlib.pyplot as plt
  mapping = {}
  for i in np.unique(y):
    mapping[i] = np.where(y == i)[0]
  l = len(mapping) #Numero de clases
  if titles == None:
    titles = ["Unknown"]*l
  images = []
  for j in range(l):
    images.append(X[mapping[j][n_image]])
```

```
  for k, image in enumerate(images):
    plt.subplot(len(images) / columns + 1, columns, k + 1)
    plt.axis('off')
    plt.title(str(k) + ' ' + titles[k])
    plt.imshow(image)
```

We execute the function by passing a value to the titles parameter

```
titles = ["plane", "car","bird", "cat", "deer", "dog", "frog",
    "horse", "ship", "truck"]

inspect_images(x_test, y_test, titles=titles)
```

and we get the following values:

Figure 1.1: *First image of each CIFAR10 class*

C. **Build an image that compares 10 different ship class images.**
We finished the exploratory analysis by displaying 10 random images from the *ship* class. We see how the function is valid for displaying an arbitrary number of images of any class.

```
def inspect_image(X, y, n_class, n_images=10, title = None,
    columns=5):
  import numpy as np
  import matplotlib.pyplot as plt
  import random
  if title == None:
    title = "Unknown"
  indices = np.where(y==n_class)[0]
  images = []
  for j in range(n_images):
    images.append(X[random.choice(indices)])
```

```
    for k, image in enumerate(images):
      plt.subplot(len(images) / columns + 1, columns, k + 1)
      plt.axis('off')
      plt.title(str(k))
      plt.imshow(image)
    plt.suptitle(title)
```

If we execute the function for the class *n_class* = 8 corresponding to the ships for the test set, we get the following results

Figure 1.2: *Random images of the ship class*

2 Training a customized CNN

We continue the practice by training a custom CNN provided by the subject for the data set shown above. Examining this code briefly, we see that it employs a convolutional neurak network composed of 10 layers. These layers are:

- Four convolutional layers , with a kernel size (5×5).
- Two layers of *MaxPooling2D* pooling.
- Three layers of dropout.
- A dense layer of 512 neurons.
- A dense layer with 10 neurons (the number of classes) and softmax activation to determine the final result

The implementation is done on TensorFlow's Keras library on GPUs.

A. **Entrenar el modelo para un kernel de tamaño** (5×5) **y otro de tamaño** (3×3) **y presentar los resultados.**

   We start by running the code provided with the parameters *batch_size=32* and *epochs=2* with the data enhancement function disabled. Once the model has been trained, we can use the *model.summary()* function to obtain a report of the model obtained. The results are:

Layer (type)	Output Shape	Param #
conv2d_21 (Conv2D)	(None, 32, 32, 32)	896
activation_31 (Activation)	(None, 32, 32, 32)	0
conv2d_22 (Conv2D)	(None, 30, 30, 32)	9248
activation_32 (Activation)	(None, 30, 30, 32)	0
max_pooling2d_11 (MaxPooling	(None, 15, 15, 32)	0
dropout_16 (Dropout)	(None, 15, 15, 32)	0
conv2d_23 (Conv2D)	(None, 15, 15, 64)	18496
activation_33 (Activation)	(None, 15, 15, 64)	0
conv2d_24 (Conv2D)	(None, 13, 13, 64)	36928
activation_34 (Activation)	(None, 13, 13, 64)	0
max_pooling2d_12 (MaxPooling	(None, 6, 6, 64)	0
dropout_17 (Dropout)	(None, 6, 6, 64)	0
flatten_6 (Flatten)	(None, 2304)	0
dense_11 (Dense)	(None, 512)	1180160
activation_35 (Activation)	(None, 512)	0
dropout_18 (Dropout)	(None, 512)	0
dense_12 (Dense)	(None, 10)	5130
activation_36 (Activation)	(None, 10)	0

Total params: 1,250,858
Trainable params: 1,250,858
Non-trainable params: 0

Figure 2.1: *Summary for the model using* 3×3 *convolutional kernels*

Layer (type)	Output Shape	Param #
conv2d_25 (Conv2D)	(None, 32, 32, 32)	2432
activation_37 (Activation)	(None, 32, 32, 32)	0
conv2d_26 (Conv2D)	(None, 28, 28, 32)	25632
activation_38 (Activation)	(None, 28, 28, 32)	0
max_pooling2d_13 (MaxPooling	(None, 14, 14, 32)	0
dropout_19 (Dropout)	(None, 14, 14, 32)	0
conv2d_27 (Conv2D)	(None, 14, 14, 64)	51264
activation_39 (Activation)	(None, 14, 14, 64)	0
conv2d_28 (Conv2D)	(None, 10, 10, 64)	102464
activation_40 (Activation)	(None, 10, 10, 64)	0
max_pooling2d_14 (MaxPooling	(None, 5, 5, 64)	0
dropout_20 (Dropout)	(None, 5, 5, 64)	0
flatten_7 (Flatten)	(None, 1600)	0
dense_13 (Dense)	(None, 512)	819712
activation_41 (Activation)	(None, 512)	0
dropout_21 (Dropout)	(None, 512)	0
dense_14 (Dense)	(None, 10)	5130
activation_42 (Activation)	(None, 10)	0

Total params: 1,006,634
Trainable params: 1,006,634
Non-trainable params: 0

Figure 2.2: *Summary for the model using* 5×5 *convolutional kernels*

B. **Check that the number of parameters of the convolutional layers**
We see how we get 1,250,858 trainable parameters for the model (3×3) and 1,006,634 for the model (5×5). This is surprising since at first glance it might seem that the (5×5) model would contain more parameters due to the larger size of the kernels. However, this is because the largest number of parameters are contained in the dense layer and this depends on the number of neurons in the previous layer. Larger convolutional kernels decrease the sizes of the subsequent convolutional layers more rapidly, and for this reason the number of parameters ends up being smaller. We can see that for the (3×3) model the number of neurons that reach the dense layer is 2,304, while for the $(5times5)$ model it is 1,600.

We can obtain the number of parameters of the model by doing a theoretical analysis, since this is given by the number of parameters of the kernel and its size. So, if we consider that a convolutional layer with N cores of $m \times m$ has

$$param_{conv} = N \cdot m^2 \cdot NC + N$$

parameters, where NC is the number of channels; and a dense layer has

$$param_{dense} = N \cdot (M + 1))$$

where N is the number of neurons in the layer, M is the number of neurons in the previous layer and the term +1 corresponds to the bias. Thus, it is easy to check that the numbers provided by the report are correct.

C. **Report the CPU time needed to train the model.**
We continue to study the model's execution time by varying the different parameters. In particular, we study the accuracy of the model (3×3) and the execution time running the model on GPU as well as CPU, varying the data_augmentation parameter and the number of epochs. The results are the following:

Epochs	Test accuracy	Elapsed time (s)	Device	Data augmentation
2	51,05%	28	GPU	NO
2	48,44%	35	GPU	YES
2	49,86%	362	CPU	NO
2	50,62%	368	CPU	YES
10	67,95%	134	GPU	NO
10	64,19%	239	GPU	YES
10	71,19%	2.292	CPU	NO
10	69,40%	1.638	CPU	YES

We see how the number of epochs has a decisive influence on the quality of the model, which is to be expected. In turn, we see how the data_augmentation parameter decreases the accuracy of the model and also increases the training time, so for this data set it does not seem to be beneficial. Finally, we see how GPU training results in an acceleration of training time by a factor of approximately $\times 10$. The GPU used is an NVIDIA GTX 1060 3GB.

D. **Evaluate the overall accuracy of the model.**
We study the accuracy of both models from the accuracy matrices and sklearn classification reports. We start by defining the precision, recall and f1-score magnitudes.

Accuracy is defined as the ratio between the number of accurate predictions and the total number of predictions made for a class, while recall or sensitivity measures the number of accurate predictions among the total number of individuals in a class.

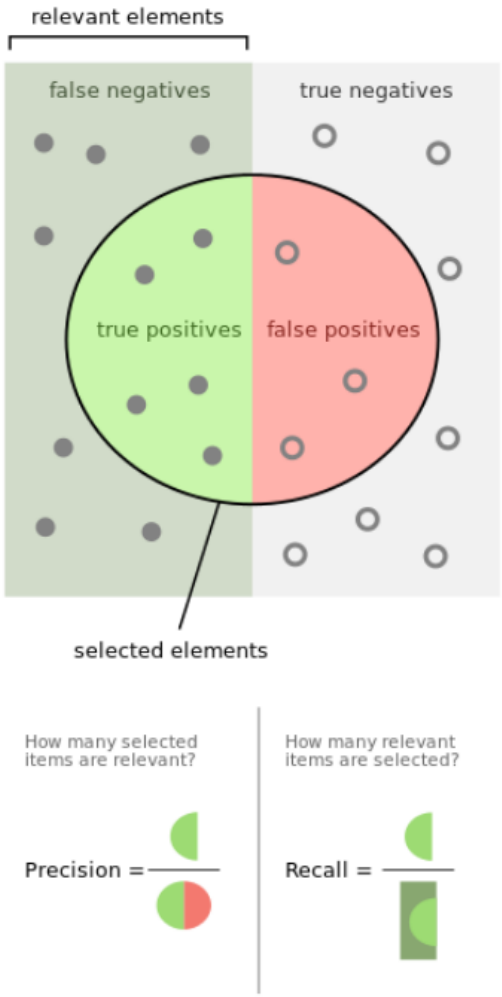

Figure 2.3: *Representation of precision and recall for a given class*

Finally, the f-score is defined as

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

The confusion matrix can be obtained by an sklearn method, and all these previous magnitudes can be obtained by the report of *sklearn*, once the predictions of the model are obtained. To do this, we execute the code:

```
import sklearn
import seaborn as sn

#Confusion matrix using Seaborn
cm = sklearn.metrics.confusion_matrix(y_test_2, y_predict_2)
sn.heatmap(cm, annot=True, fmt='g')

#Sklearn report of precision, recall and F1-score
sklearn.metrics.classification_report(y_true, y_pred)
```

The results are:

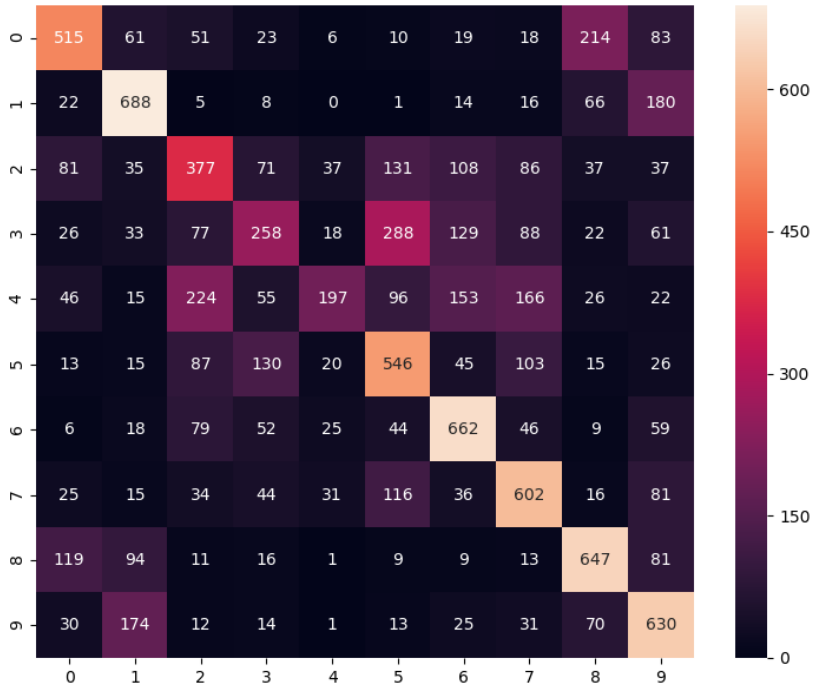

Figure 2.4: *Confusion matrix for the* 3×3 *model*

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8	Class 9	Average
Precision	0.56	0.67	0.49	0.38	0.53	0.51	0.47	0.48	0.68	0.55	0.53
Recall	0.61	0.65	0.32	0.36	0.28	0.37	0.80	0.64	0.58	0.67	0.53
F1-score	0.59	0.66	0.39	0.37	0.36	0.43	0.59	0.55	0.63	0.61	0.52

Table 2.1: *Classification report for the* 3×3 *model*

We see how for the model 3×3 for 2 periods we obtain values of precision between 0.5 and 0.6 for the different classes. Observing the confusion matrix, we see that the classes {2,3,4,5} present a greater degree of confusion than the rest of the classes (a smaller *recall*). In particular, there is a high degree of confusion between classes 2, 4 (bird, deer) and between classes 3, 5 (cat, doog) which makes sense since they are images with a high degree of similarity to each other.

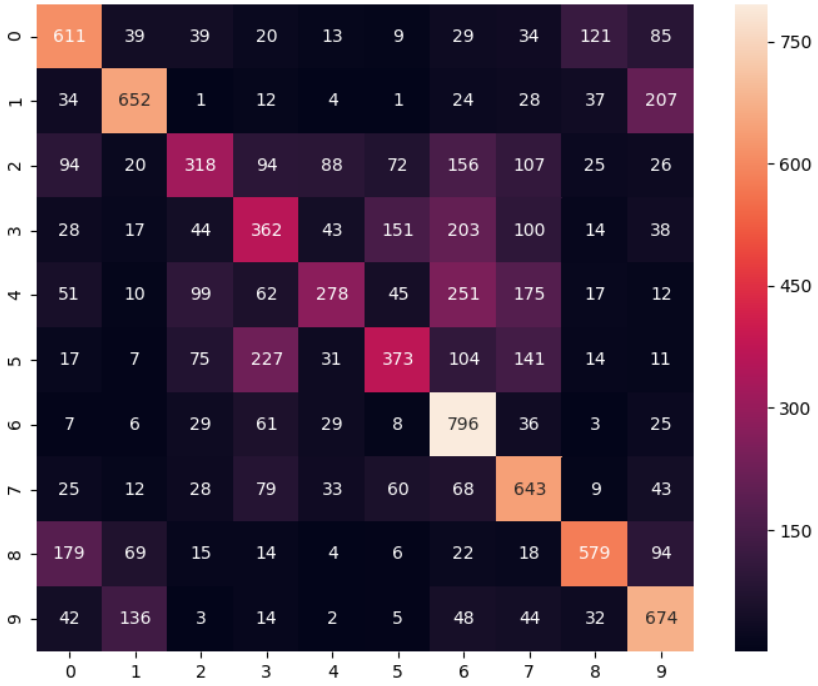

Figure 2.5: *Confusion matrix for the* 5×5 *model*

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8	Class 9	Average
precision	0.58	0.60	0.39	0.38	0.59	0.44	0.55	0.51	0.58	0.50	0.51
recall	0.52	0.69	0.38	0.26	0.20	0.55	0.66	0.60	0.65	0.63	0.51
f1-score	0.55	0.64	0.39	0.31	0.29	0.48	0.60	0.56	0.61	0.56	0.50

Table 2.2: *Classification report for the* 5×5 *model*

Just as for the 5×5 model, we see that classes $\{2, 3, 4, 5, 6\}$ present a greater degree of confusion than the rest, mainly classes 3 and 4 because of their low degree of recall. Some classes are slightly confused, such as the car class with the truck class, or the plane class with the ship class.

3 Training a predefined CNN (VGG16)

A. **Create a model using the VGG16 architecture and provide an overview of the resulting architecture.**
We perform a similar procedure to the previous one but this time using the VGG16 architecture. This architecture composes a CNN formed by 13 ReLU convolutional layers, 5 pooling layers and 4 dense layers.

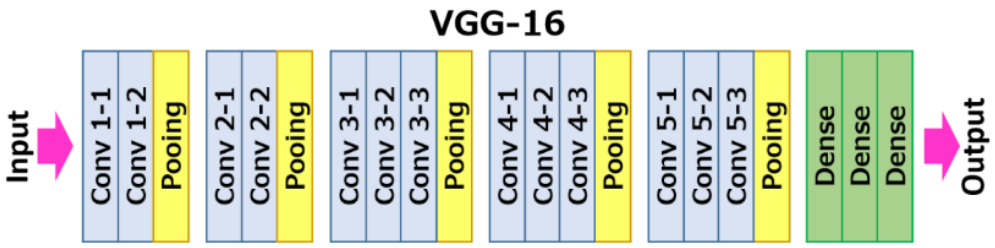

Figure 3.1: Representation of the VGG16 architecture

the main differences with respect to the architecture in the previous section are the greater number of both convolution and pooling layers and densities. The similarities can be found in the kernel sizes for the convolutional layers, which are 3×3 as well as in the ReLU type activation functions.

B. **Train the model VGG16**
We proceed to train this CNN on CIFAR10 data for the values epochs = 2 and epochs = 10 in order to compare the results with the previous architecture. By doing this, we get:

Epochs	Test accuracy	Elapsed time (s)
2	48,76%	183
10	88,78%	911

We see how the precision of the model on the test set improves in both cases with respect to the previous architecture, especially for the case with 10 periods, although it also increases considerably the training time. In fact, one of the biggest disadvantages of the VGGNet architecture is the long training time.

Below is a graphic comparison between the two architectures:

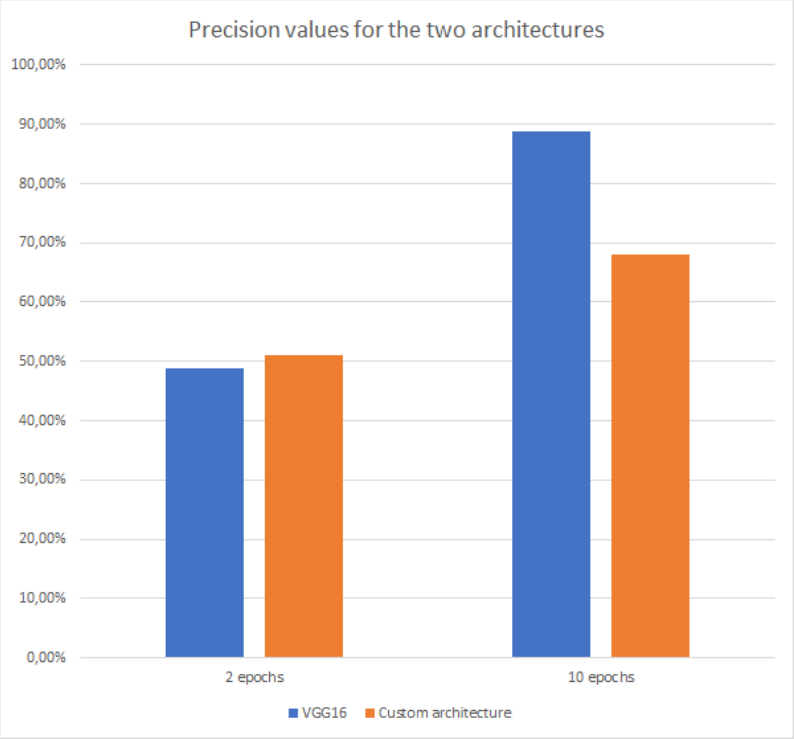

Figure 3.2: *Representation of the VGG16 architecture*

Figure 3.3: *Representation of the VGG16 architecture*

We see how the improvement in accuracy of the VGG network (around 20% for 10 seasons) is offset by a considerable increase in training time by a factor of 9×9.

C. **Apply the VGG16 previously trained with the ImageNet dataset**
We proceed to observe a transferred learning process using the VGG16 architecture trained on the ImageNet dataset. To do this, we have done the following. First, we created a new model with vgg16 architecture trained with the imagenet weights.

```
model = VGG16(include_top=False, weights='imagenet')
```

the result is a matrix of 512 numbers for each image in the data set. This is because the last layer of vgg16 is a dense layer of 512 neurons, and each of these numbers corresponds to a feature that extracts the model from the images. That is, this model acts as a feature extractor, based on the features learned by CNN from ImageNet.

Then, we performed a feature selection process on these 512 features extracted by vgg16. We do this because a lot of these features are zeros, and very possibly keeping a small number of *k* features will improve the performance of the model when giving the data to the SVM.

There is a wide variety of feature selection algorithms, and this forms a field of its own within machine learning. For simplicity, we use sklearn's feature selector *SelectKBest* and keep the top 100 features using a chi-square filter. It is important to note that we train the selector only on the training set since we want to remove the same features from the training set and the test set.

```
def select_features(features, labels):
  from sklearn.feature_selection import SelectKBest, chi2
  selector = SelectKBest(chi2, k=100)
  selector.fit(features, labels)
  return selector
```

Finally, we train a linear kernel SVM with these features obtained from CIFAR10 training data, and we calculate the score obtained on the CIFAR10 test set following the same procedure. To calculate the score on the test set, we retrieve the code provided for the CET2 about cross-validation:

```
features = process_features_vgg16(x_train)
selector = select_features(process_features_vgg16(x_train), y_train)
x_train = selector.transform(process_features_vgg16(x_train))
x_test = selector.transform(process_features_vgg16(x_test))

SVM = LinearSVC(C = 0.025)
SVM.fit(x_train, y_train)

cv_results = cross_validate(SVM, x_test, y_test, cv=5)
test_score = cv_results['test_score']
fit_time = cv_results['fit_time']
score_time = cv_results['score_time']
```

The results of this procedure are as follows:

	Test score	Score time (s)	Fit time (s)
Mean	0,5150	0,0016	6,3888
Standard deviation	0,0081	0,0005	0,0539

Comparing it with the previous results:

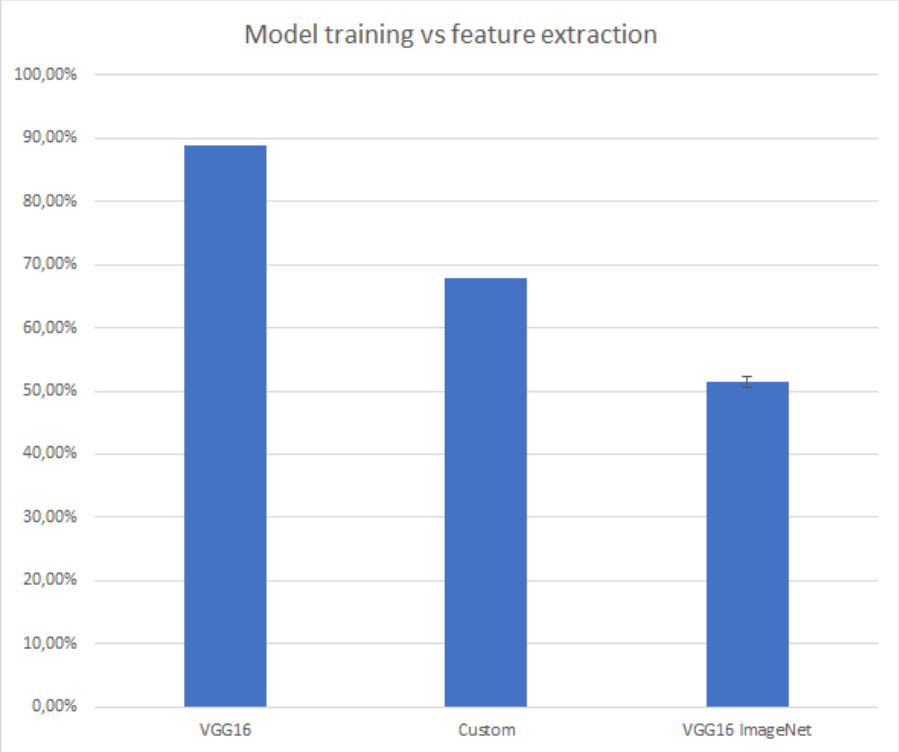

Figure 3.4: *Scores for the previous models vs. score for pretrained VGG16 on ImageNet*

It is important to see that, although by doing this we have saved ourselves from training the CNN from scratch with the CIFAR10 data, it is a computationally demanding process since the calculation of the features from the pre-trained vgg16 model takes time.

We see how the score from feature extraction is worse than the rest. However, a score of 50% when the model has never seen the images from CIFAR10 and has trained from a totally different set of images is a quite surprising result.

4 Visualization of the map of characteristics

We ended the practice by studying the feature maps formed by the CNN when processing an image of the CIFAR10 array. In particular, we studied the feature maps of the first image of the dataset, namely

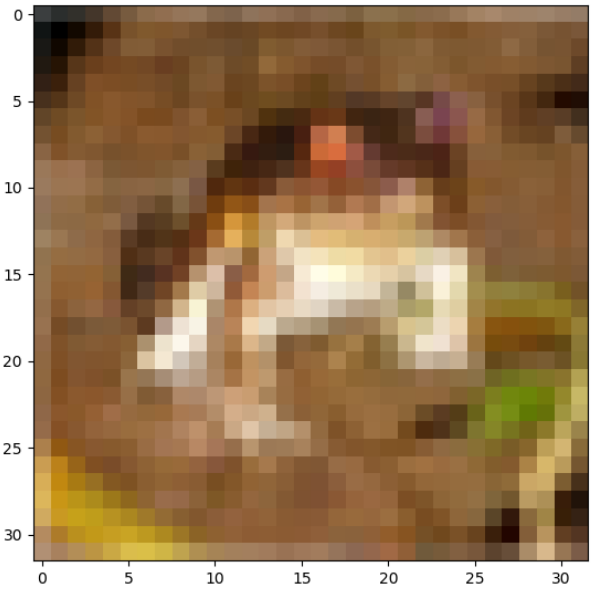

Figure 4.1: *First image of the CIFAR10 dataset. Class: frog*

This is a frog class image.

A. **For the models in exercise 2 with** 5×5 **and** 3×3 **convolutional layers, extract the feature maps of the first convolutional layer**
To calculate the feature maps of a CNN, what we want to do is to pass as input an image only to one layer of the CNN and represent the output as an image. To do this, we create a new model with Keras' *Model()* method, passing the input and output parameters in such a way that we only study one layer of the CNN:

```
model_2 = Model(inputs=model.input,
    outputs=model.get_layer('layer_name').output)
```

In this way, by means of the method *model_2.predict()* we can study the response of a layer of the CNN to the image. All this can be summarised in the following function, which calculates the characteristics map of the n-th image in CIFAR10:

```
def characteristic_map(model, layer_name, image_n=0):
  import numpy as np
  from keras.models import Model
  model_2 = Model(inputs=model.input,
      outputs=model.get_layer(layer_name).output)
  img = x_train[image_n]
  img = np.expand_dims(img, axis=0)
  conv2d_features = model_2.predict(img)
  print(conv2d_features.shape)

  import matplotlib.pyplot as plt
  square = 8
  ix = 1
  for _ in range(square):
  for _ in range(square):
  ax = plt.subplot(square, square, ix)
  ax.set_xticks([])
  ax.set_yticks([])
  plt.imshow(conv2d_features[0, :, :, ix-1], cmap='gray')
  ix += 1
  plt.show()
```

Using this function for the custom architecture of exercise 2, for the kernels of size 3×3 and 5×5 we obtain the following images:

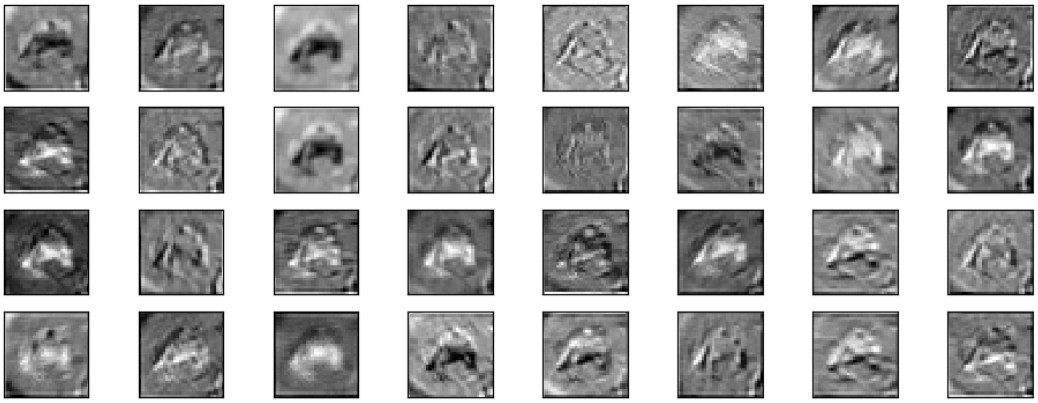

Figure 4.2: *Feature map for the* 3×3 *custom CNN architecture*

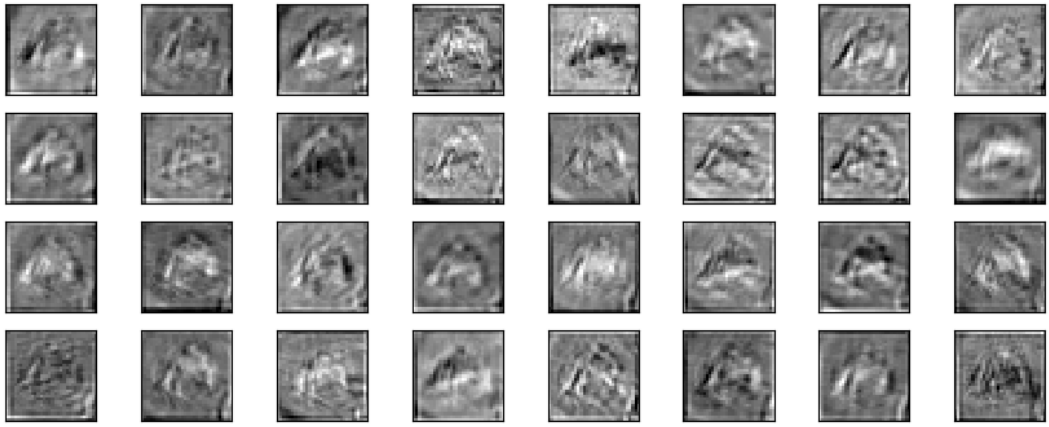

Figure 4.3: *Feature map for the* 5×5 *custom CNN architecture*

We see how the layer with a 3×3 core produces slightly sharper images than for the larger core, i.e. it contains more information. We see how we can easily see the silhouette of the frog. This means that by training the model with the entire data set, it has learned to extract silhouettes.

B. **Compare the feature maps of the first convolutional layer of the pre-trained VGG16 and the non-trained VGG16** We proceed to apply the same procedure but this time to the VGG16 models trained for CIFAR10 data and pretrained with ImageNet. This time we keep the first 64 features to make them easier to visualize. The results are the following:

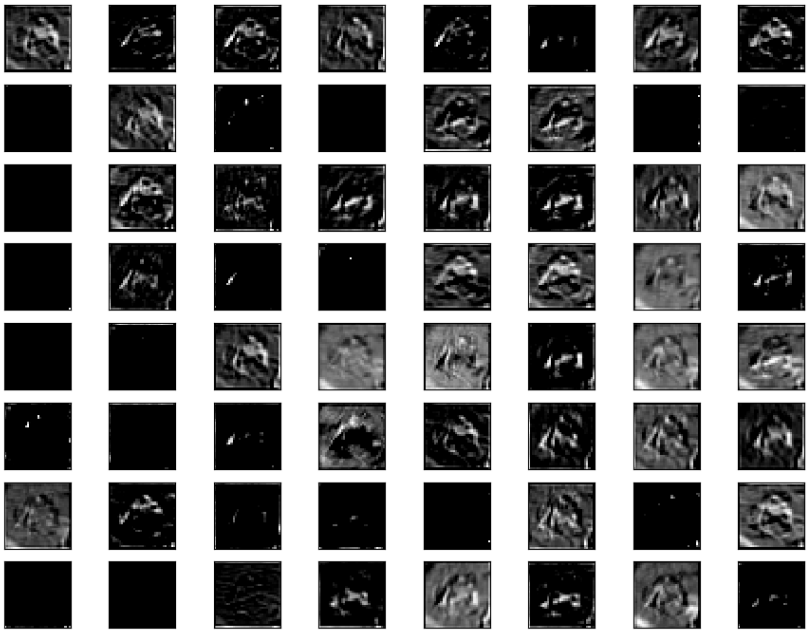

Figure 4.4: *Feature map for the CIFAR10 VGG16 model*

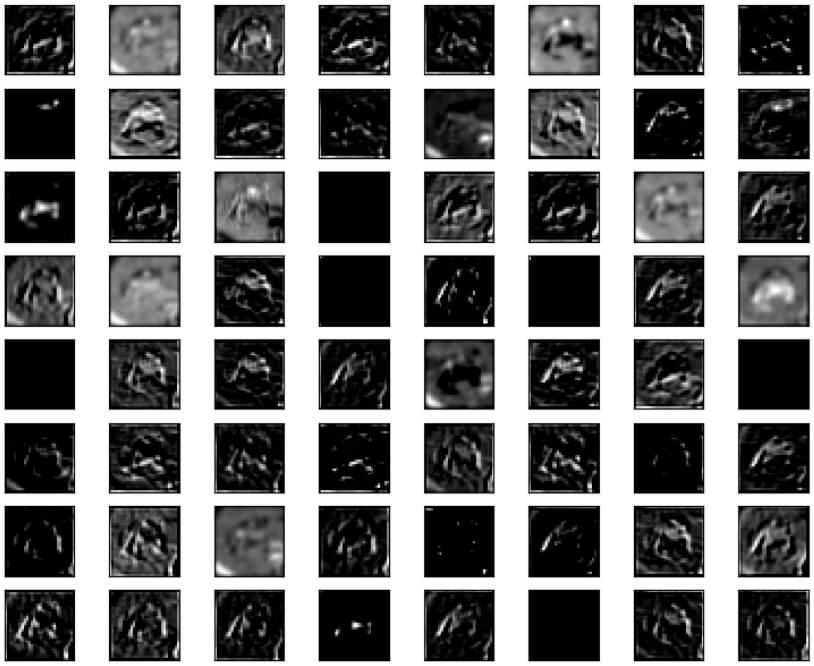

Figure 4.5: *Feature map for the ImageNet pre-trained VGG16 model*

We see how the results are very similar to each other. The contrast obtained by the VGG16 model is greater than for the previous case, and the features extracted by the network are much more concrete than for the previous case. It is difficult to draw conclusions and differences from the images, but we can appreciate how for the untrained case, there are more features that are totally black at first, while for the pretrained case it seems that the network is able to differentiate more features for the first network.

C. **Compare the feature maps of the first and last convolutional layer of the pre-trained VGG16.**
We repeat this procedure for the last time this time for the first and last layer of the VGG16 model pre-trained with ImageNet data.

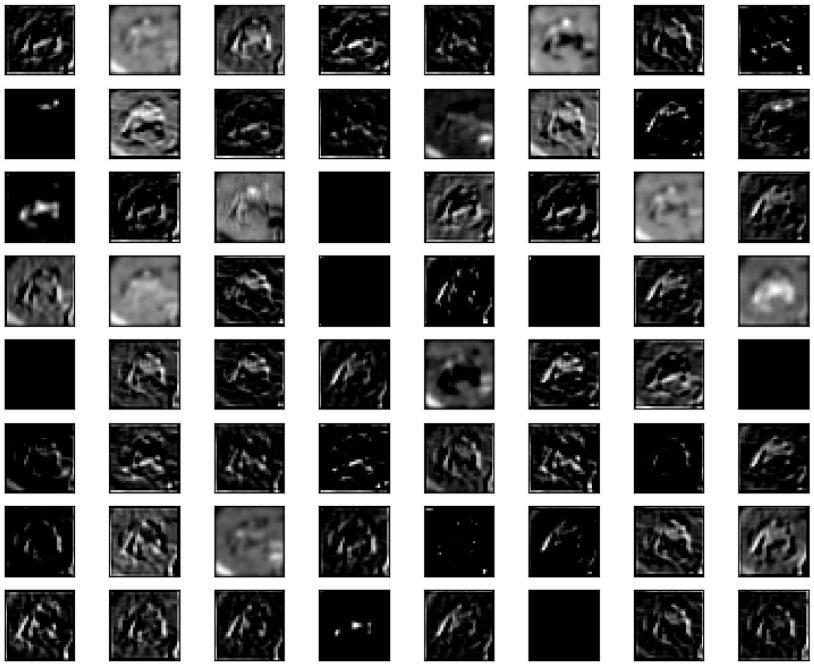

Figure 4.6: *Feature map for the first layer of ImageNet pre-trained VGG16 model*

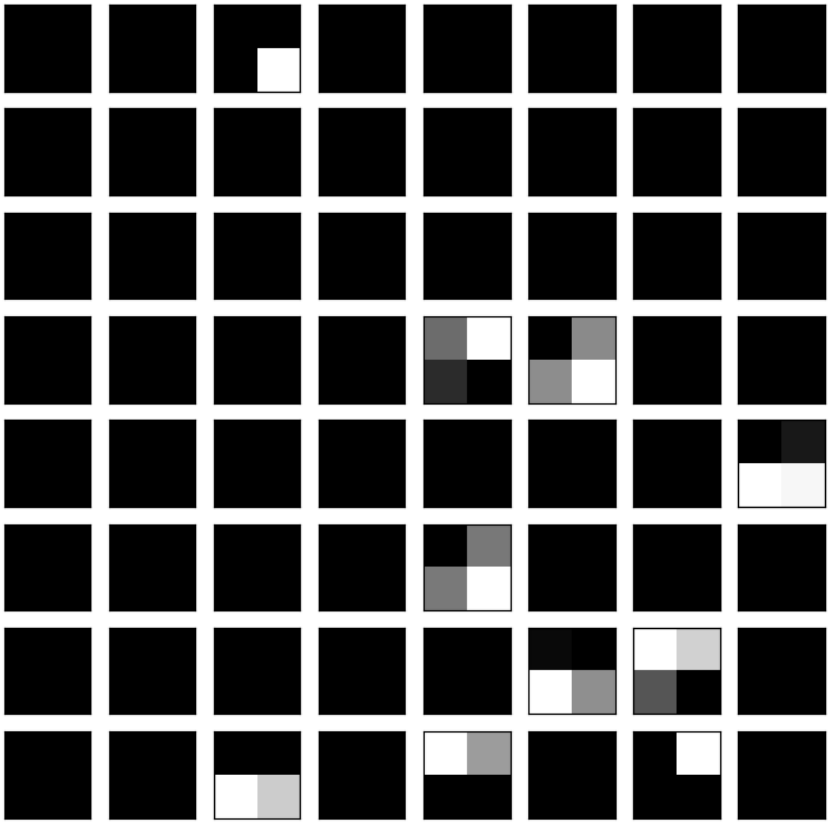

Figure 4.7: *Feature map for the last layer of ImageNet pre-trained VGG16 model*

We see how the first layer captures much more information about the images than the last one. This makes sense since the objective of CNN is to pass information to a multi-layer perception at the end of the layer to classify the images, and to do this the information is processed more and more until at the end you get a list of 512 numbers that are passed to the perception. In a way, each layer extracts features from the previous layer and therefore it makes sense that each layer contains less information than the previous one.