Assignment 5

Deep Learning CNN

Gonçalo Bernardo

Table of contents

Index of images

Index of tables

1. Project description

The present project pretends to use Machine Learning techniques to classify images from the CIFAR10 dataset included in Keras, compute metrics as loss, accuracy from test and train datasets, and analyse the results in terms of over/underfitting and efficiency.

2. Project objectives

The main goal of the project is to test some different combinations of elements (Networks architecture, data augmentation…) and to perform a global comparison among all the combinations.

3. Project Approach

Due to the nature of the project, and the need of a large number of test cases, the start point was to test with some simple basic approaches, with no techniques as data augmentation or early stopping.
Here, in the report, it will be shown only tests that are relevant and interesting to show the iterative process, the adjustments and continuous improvements.

4. Custom Architecture Models' description

4.1 First Model

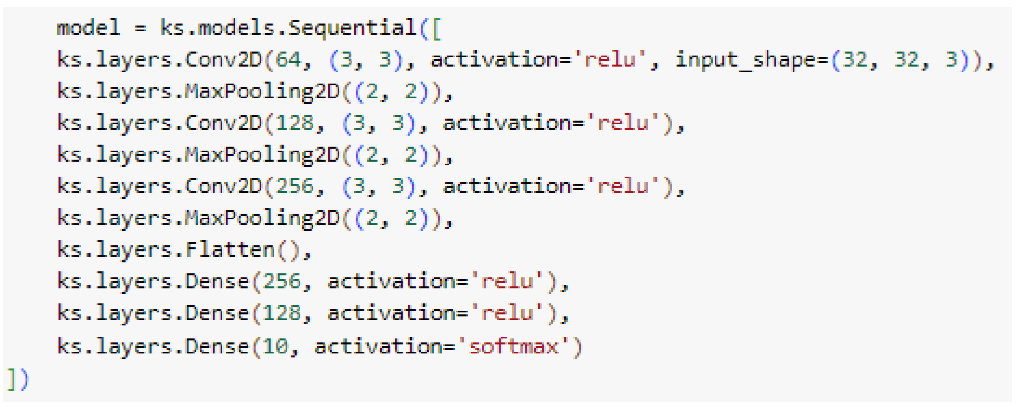
To start the process, apart from the default parameters' values, there were considerate the following parameters:

Table 4.1.1 - Parameters considered for the first model.

Architecture	Not specified.
Convolution Layers	3 layers, with 64, 128 and 256 filters (by that order) and all with Rectified Linear Unit activation function.
MaxPooling2D*	(2,2) - after each of convolution layer
Dense layers	2 layers with 256 and 128 filters, (by that order) and all with Rectified Linear Unit activation function.
Dropout	No dropout.
Output layer	10 filters and Normalised exponential (softmax) activation function.
Optimizer	Adam Optimization Algorithm
Loss function	Sparse Categorical Cross Entropy

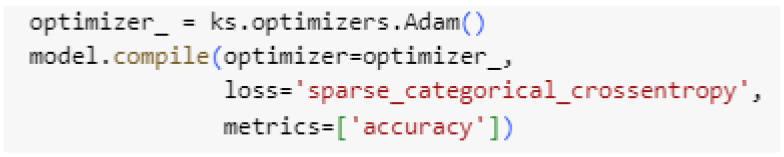
*It helps in reducing the spatial dimensions, which reduces the number of parameters in the model and helps in controlling overfitting. Typically, MaxPooling2D is used after every one or two convolutional layers

There were considered a total of 40 epochs to the training, and no early stopping.
Below one can see some of the relevant parts of the code implementing that model:



```
    model = ks.models.Sequential([
    ks.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    ks.layers.MaxPooling2D((2, 2)),
    ks.layers.Conv2D(128, (3, 3), activation='relu'),
    ks.layers.MaxPooling2D((2, 2)),
    ks.layers.Conv2D(256, (3, 3), activation='relu'),
    ks.layers.MaxPooling2D((2, 2)),
    ks.layers.Flatten(),
    ks.layers.Dense(256, activation='relu'),
    ks.layers.Dense(128, activation='relu'),
    ks.layers.Dense(10, activation='softmax')
])
```

Fig 4.1.1 - Model definition (1st model).



```
optimizer_ = ks.optimizers.Adam()
model.compile(optimizer=optimizer_,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Fig 4.1.2 - Definition of optimizer and loss function.

4.1.1 Results

Table 4.1.2 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
97.85%	71.84%	70.87%

Table 4.1.3 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.066	2.2122

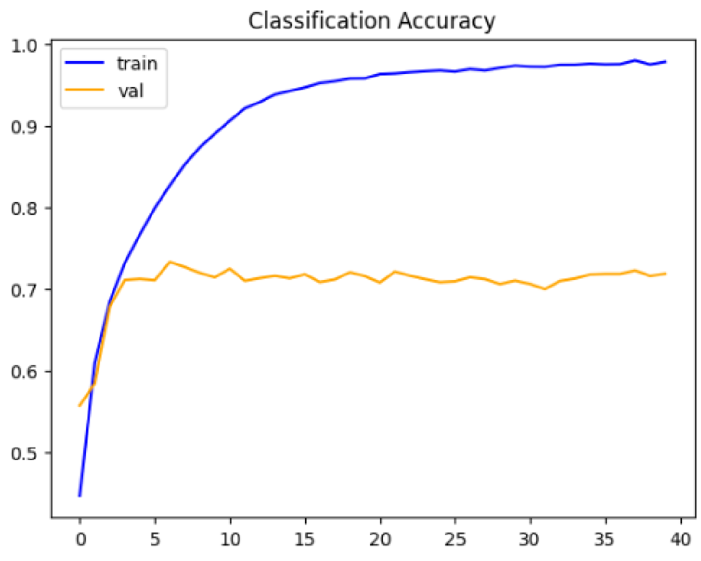
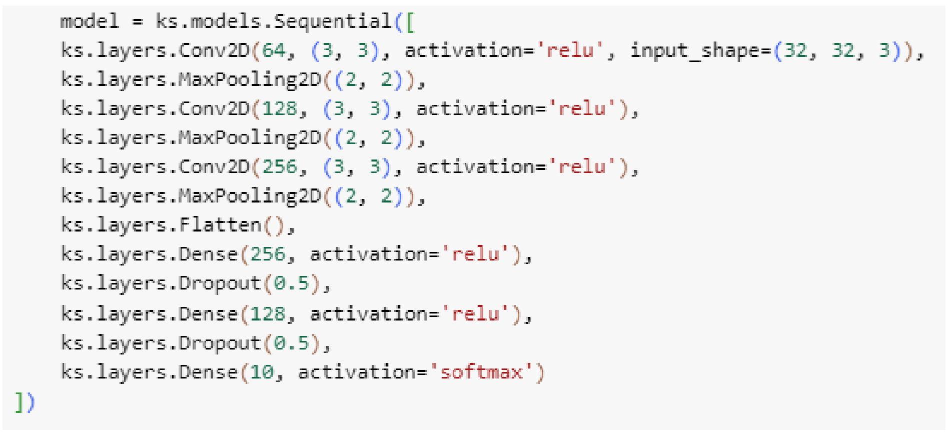


Fig 4.1.3 - Accuracy of train dataset (blue) and validation dataset (orange).

4.2 Second Model

The parameters used were identical to the previous model, except that a dropout was used after the two dense layers, as we can see in figure 4.2.1. Also it was considered an early stopping using as monitor metric "validation loss” and “validation accuracy” and for both a patience equals to 10. The epochs were set to 100.



```
    model = ks.models.Sequential([
    ks.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    ks.layers.MaxPooling2D((2, 2)),
    ks.layers.Conv2D(128, (3, 3), activation='relu'),
    ks.layers.MaxPooling2D((2, 2)),
    ks.layers.Conv2D(256, (3, 3), activation='relu'),
    ks.layers.MaxPooling2D((2, 2)),
    ks.layers.Flatten(),
    ks.layers.Dense(256, activation='relu'),
    ks.layers.Dropout(0.5),
    ks.layers.Dense(128, activation='relu'),
    ks.layers.Dropout(0.5),
    ks.layers.Dense(10, activation='softmax')
])
```

Fig 4.2.1 - Model definition (2nd model).

4.2.1 Results

Table 4.2.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
85.01%	70.68%	70.52%

Table 4.2.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.4413	1.1248

Training stopped at epoch 18.

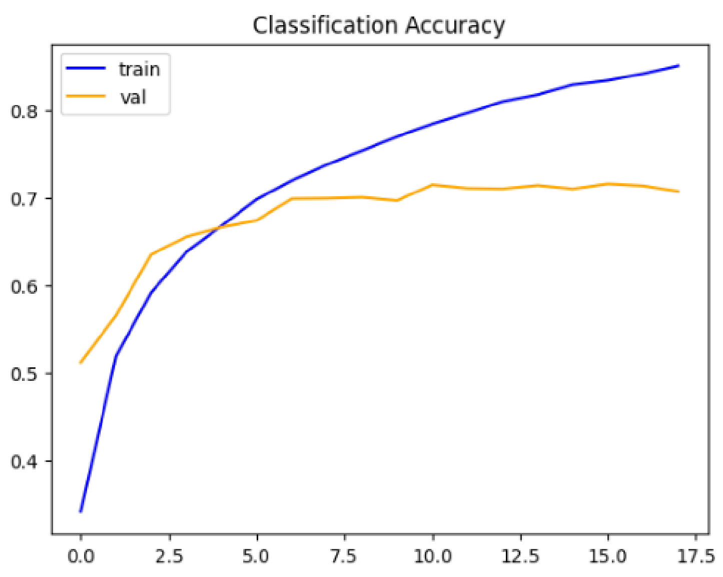
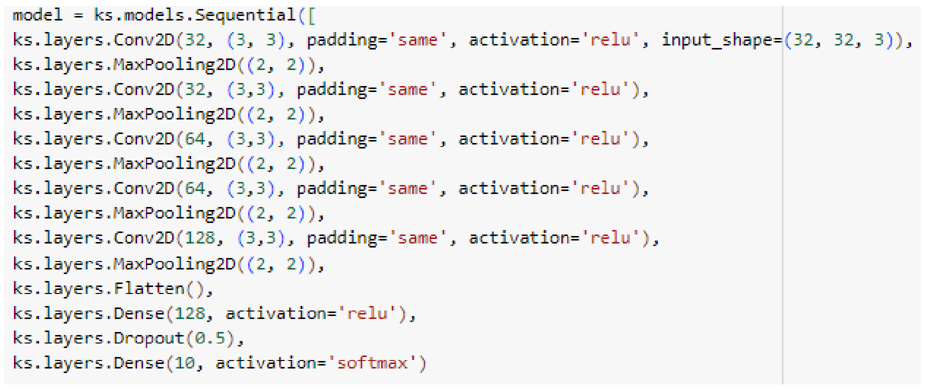


Fig 4.2.2 - Accuracy of train dataset (blue) and validation dataset (orange).

4.3 Third Model

After several trials, more 2 convolution layers were added, which makes it necessary to use padding to align input and output shapes, and don't have conflicts in the spatial dimensions. Also the complexity of the model was reduced (less filters) and one dense layer was removed. See figure 4.3.1.



```
model = ks.models.Sequential([
ks.layers.Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Conv2D(32, (3,3), padding='same', activation='relu'),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Conv2D(64, (3,3), padding='same', activation='relu'),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Conv2D(64, (3,3), padding='same', activation='relu'),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Conv2D(128, (3,3), padding='same', activation='relu'),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Flatten(),
ks.layers.Dense(128, activation='relu'),
ks.layers.Dropout(0.5),
ks.layers.Dense(10, activation='softmax')
```

Fig 4.3.1 - Model definition (3nd model).

4.3.1 Results

Table 4.3.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
83.11%	70.42%	69.06%

Table 4.3.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.4721	1.0302

Training stopped at epoch 16.

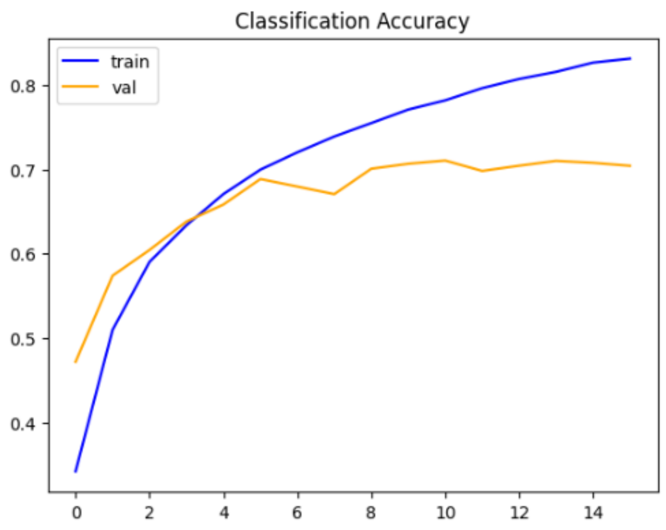
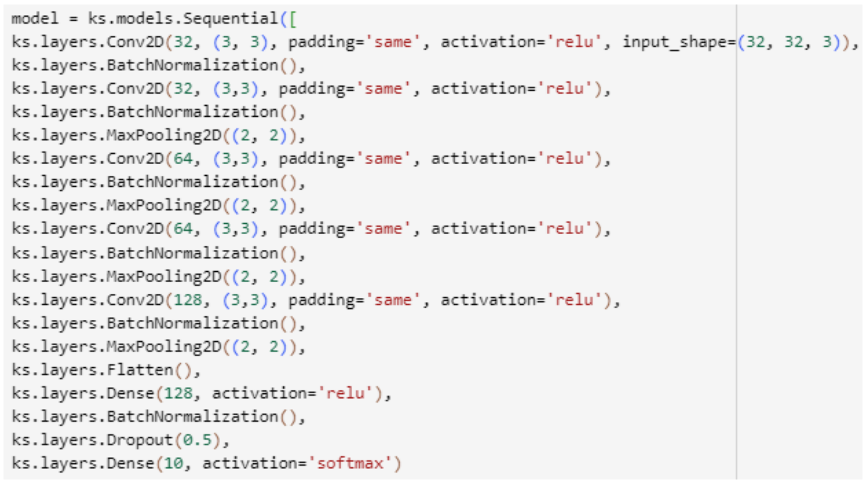


Fig 4.3.2 - Accuracy of train dataset (blue) and validation dataset (orange).

4.4 Forth Model

After the previous model and a set of new trials, in order to improve the generalisation and reduce overfitting it was inserted batch normalisation layers between two consecutive layers. It consists of normalising activation vectors from layers using the first and the second statistical moments (mean and variance) of the current batch. It also enables the use of higher learning rates. Figure 4.4.1 shows the sequence of the layers in this model.



```
model = ks.models.Sequential([
ks.layers.Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)),
ks.layers.BatchNormalization(),
ks.layers.Conv2D(32, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Conv2D(64, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Conv2D(64, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Conv2D(128, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Flatten(),
ks.layers.Dense(128, activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.Dropout(0.5),
ks.layers.Dense(10, activation='softmax')
```

Fig 4.4.1 - Model definition (4th model).

4.4.1 Results

Table 4.4.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
94.45%	79.48%	78.16%

Table 4.4.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.1590	0.8567

Training stopped at epoch 18.

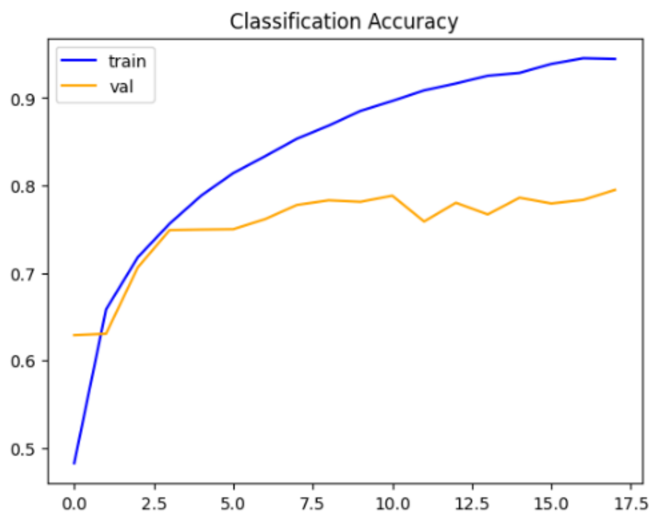
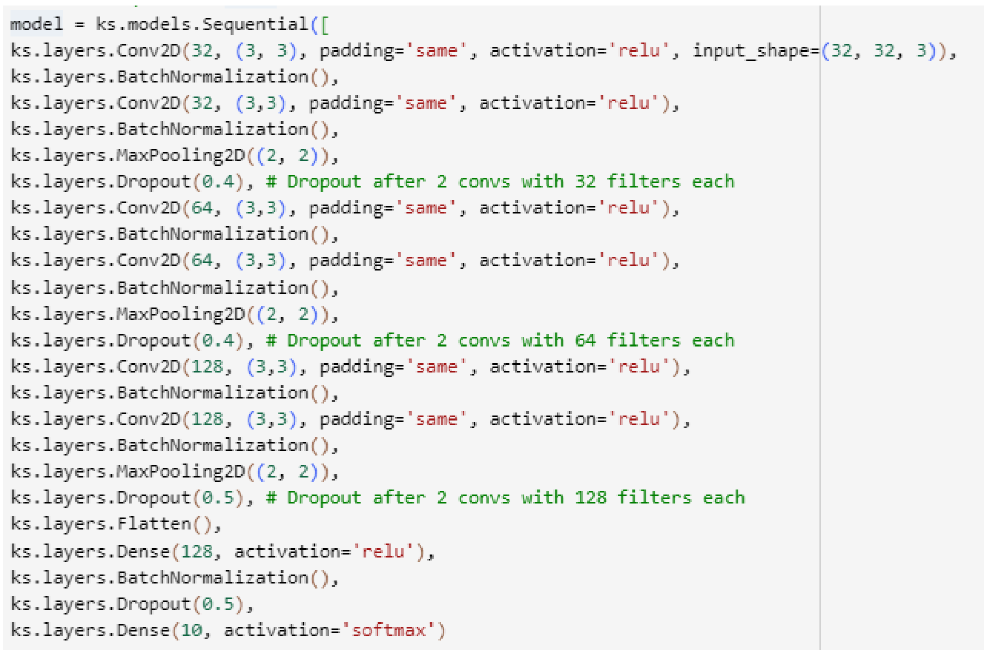


Fig 4.4.2 - Accuracy of train dataset (blue) and validation dataset (orange).

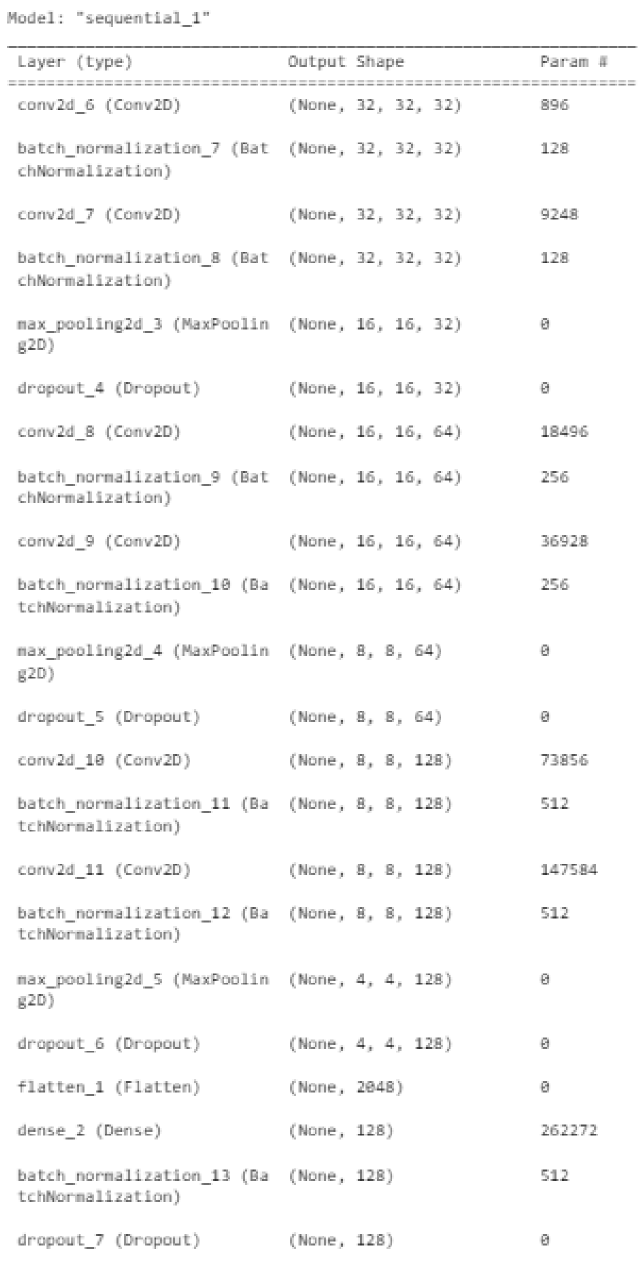
4.5 Fifth Model

After the previous model and a set of new trials, in order to improve the generalisation and reduce overfitting it was inserted batch normalisation layers between two consecutive layers. It consists of normalising activation vectors from layers using the first and the second statistical moments (mean and variance) of the current batch. It also enables the use of higher learning rates. Figure 4.4.1 shows the sequence of the layers in this model.



```
model = ks.models.Sequential([
ks.layers.Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)),
ks.layers.BatchNormalization(),
ks.layers.Conv2D(32, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Dropout(0.4), # Dropout after 2 convs with 32 filters each
ks.layers.Conv2D(64, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.Conv2D(64, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Dropout(0.4), # Dropout after 2 convs with 64 filters each
ks.layers.Conv2D(128, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.Conv2D(128, (3,3), padding='same', activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.MaxPooling2D((2, 2)),
ks.layers.Dropout(0.5), # Dropout after 2 convs with 128 filters each
ks.layers.Flatten(),
ks.layers.Dense(128, activation='relu'),
ks.layers.BatchNormalization(),
ks.layers.Dropout(0.5),
ks.layers.Dense(10, activation='softmax')
```

Fig 4.5.1 - Model definition (5th model).



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 32, 32, 32)        896

 batch_normalization_7 (Bat  (None, 32, 32, 32)        128
 chNormalization)

 conv2d_7 (Conv2D)           (None, 32, 32, 32)        9248

 batch_normalization_8 (Bat  (None, 32, 32, 32)        128
 chNormalization)

 max_pooling2d_3 (MaxPoolin  (None, 16, 16, 32)        0
 g2D)

 dropout_4 (Dropout)         (None, 16, 16, 32)        0

 conv2d_8 (Conv2D)           (None, 16, 16, 64)        18496

 batch_normalization_9 (Bat  (None, 16, 16, 64)        256
 chNormalization)

 conv2d_9 (Conv2D)           (None, 16, 16, 64)        36928

 batch_normalization_10 (Ba  (None, 16, 16, 64)        256
 tchNormalization)

 max_pooling2d_4 (MaxPoolin  (None, 8, 8, 64)          0
 g2D)

 dropout_5 (Dropout)         (None, 8, 8, 64)          0

 conv2d_10 (Conv2D)          (None, 8, 8, 128)         73856

 batch_normalization_11 (Ba  (None, 8, 8, 128)         512
 tchNormalization)

 conv2d_11 (Conv2D)          (None, 8, 8, 128)         147584

 batch_normalization_12 (Ba  (None, 8, 8, 128)         512
 tchNormalization)

 max_pooling2d_5 (MaxPoolin  (None, 4, 4, 128)         0
 g2D)

 dropout_6 (Dropout)         (None, 4, 4, 128)         0

 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 128)               262272

 batch_normalization_13 (Ba  (None, 128)               512
 tchNormalization)

 dropout_7 (Dropout)         (None, 128)               0
```

Fig 4.5.2 - Summary of the model.

4.5.1 Results

Table 4.5.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
89.25%	85.93%	85.51%

Table 4.5.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.3168	0.4443

Training stopped at epoch 59.

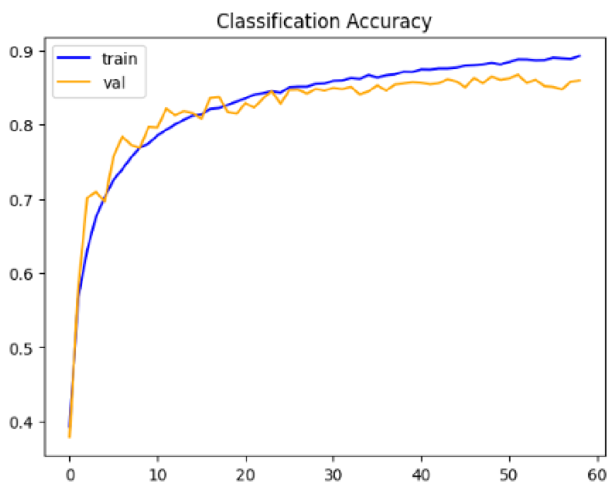


Fig 4.5.3 - Accuracy of train dataset (blue) and validation dataset (orange).

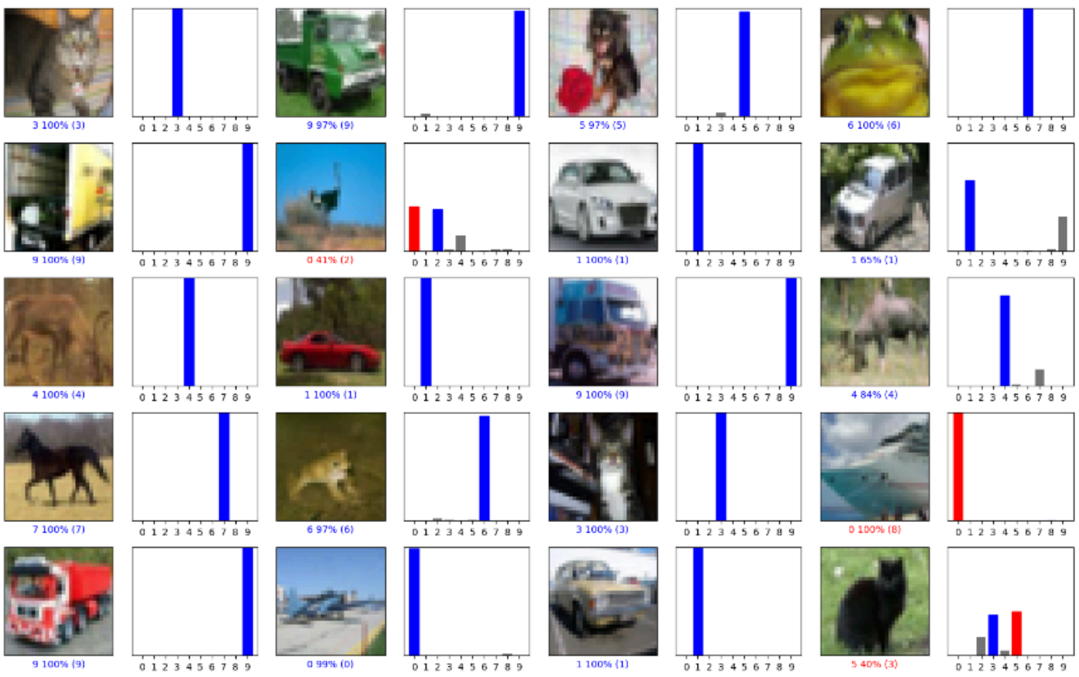


Fig 4.5.4 - Model Validation through images classification The blue colors on the graphs mean that the prediction was correct, contrarily to the red, which means that the model failed in correctly classifying the image.

4.6 Other parameters considered

4.6.1 Sigmoid function

Based on the previous model, it was considered the same parameters, but using a different activation function for the output layer: sigmoid. This function squashes the output values to the range [0,1], which can be interpreted as probabilities of belonging to the positive class. On another hand, the softmax function, used for multi-class classification, normalises the output values into a probability distribution over multiple classes, ensuring that the predicted probabilities sum up to one.
The results for this function are shown on table 4.6.1.

Table 4.6.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
85.19%	84.07%	85.46%

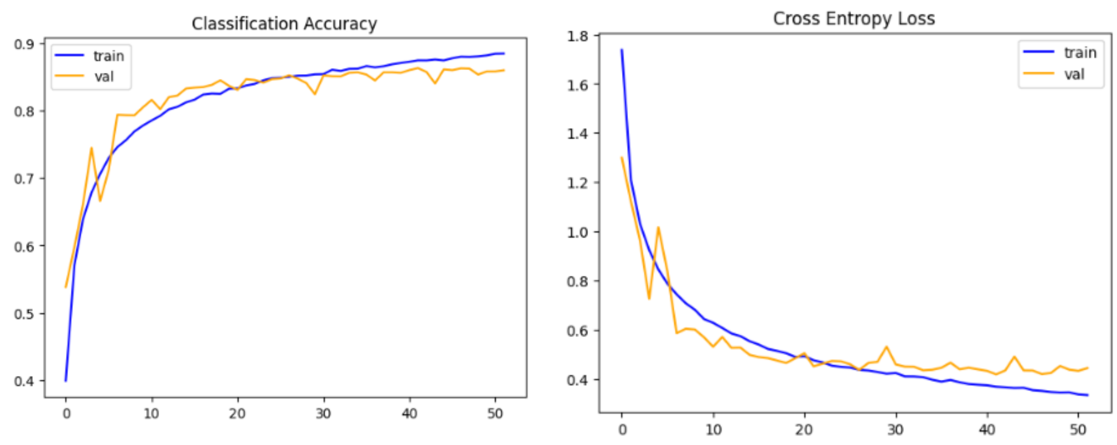
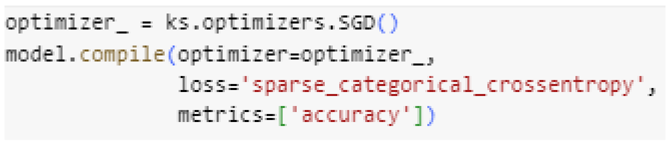


Figure 4.6.1 - (a) Accuracy of train dataset (blue) and validation dataset (orange). (b) Loss of train dataset (blue) and validation dataset (orange).

Looking at the results, one can see that the softmax function has better performance, in general, and as long as we have a multi-class classification problem (10 classes), seem to be the more appropriate choice. On another hand, looking at the results of the sigmoid function, we can see that the performance in terms of overfitting is better (better generalisation). If the primary concern is overfitting, probably it would be a better choice.

4.6.2 Stochastic Gradient Descent

Another interesting study is to analyse the effect of the optimizer function on the performance of the model during the training.
Among several options like Root Mean Square Propagation (RMSprop), Adaptive Gradient Algorithm (Adagrad), Adadelta or Stochastic Gradient Descent (SGD), it was chosen the last one which is the most basic optimization algorithm and updates the parameters based on the gradient of the loss function with respect to the parameters. Also, it can be slower to converge compared to Adam, but sometimes it can generalise better and find more optimal solutions, especially with carefully tuned learning rates.
Considering the model 4.5, it was trained with the same layers and parameters, but using SGD optimizer function.



```
optimizer_ = ks.optimizers.SGD()
model.compile(optimizer=optimizer_,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Fig. 4.6.2 - Definition of optimizer and loss function.

The results are shown on table 4.6.2.

Table 4.6.2 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
86.44%	85.67%	84.77%

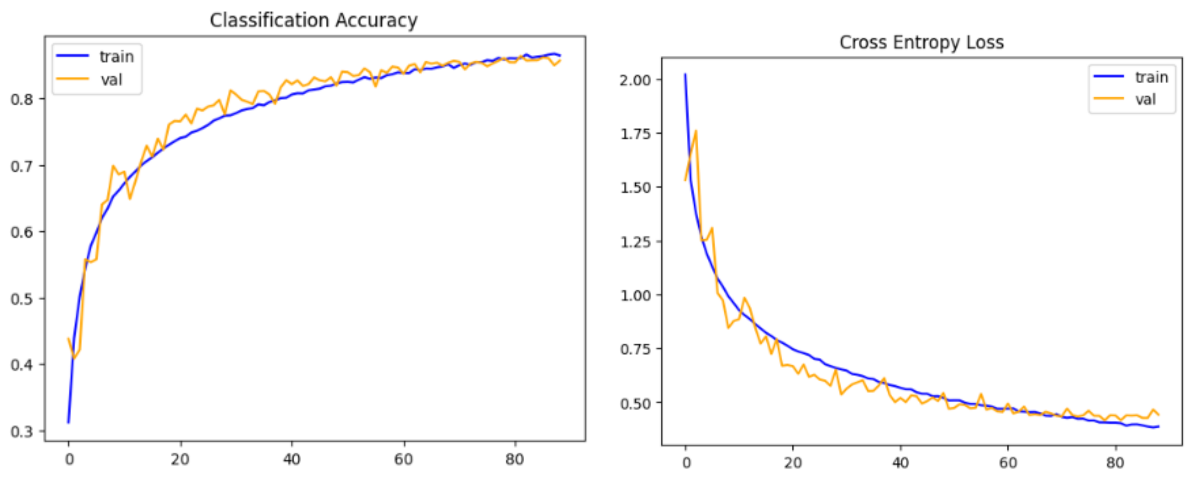


Fig 4.6.3 - (a) Accuracy of train dataset (blue) and validation dataset (orange). (b) Loss of train dataset (blue) and validation dataset (orange).

Comparing the results with results of model 4.5, one can see that, in terms of accuracy, for the train dataset, the performance was better when Adam function was used. However, for the validation and test datasets, the changes are irrelevant.

In terms of losses, the model using the SGD optimizer had strictly high values. It isn't relevant, but it could be improved by adjusting some parameters such as learning rate, momentum, initialization, and optimization hyperparameters.

4.6.3 Other parameters

To not disperse too much from the scope of the present project, other studies won't be carried out, but it would be interesting to try other types of regularisation as L2 regularisation (weight decay), which is a common technique used in neural networks to prevent overfitting. It works by adding a penalty term to the loss function during training, encouraging the model to learn simpler and smoother weight configurations. It provides smoothness and stability to the model, prevention of overfitting and controls the model's complexity.

The learning rate of optimizer (Adam and SGD) could be studied to see the effect of decreasing and increasing, which will influence the rate of convergence, stability of training or the dynamics of training, including the trajectory of the loss function and the behaviour of the model parameters over time.

Regarding the loss function used to quantify model performance and guiding model training through optimisation, the Sparse Categorical Cross Entropy function. Other functions were considered, but the performance was not satisfactory, or the code even could be compiled, so it would be interesting in future studies.

To finish, another technique, as data augmentation, could be used to increase the diversity and size of a training dataset by applying various transformations to our data samples. Those transformations could include random rotations, flips, and translations of images, changes in brightness, contrast, and saturation or zooming and cropping of images. It could improve the generalisation or the regularisation of the model.

5. Xception Architecture Models' description

The Xception model, short for "Extreme Inception," is a deep convolutional neural network architecture which represents a significant evolution of the Inception architecture, aiming to push the boundaries of performance and efficiency in image classification tasks.
Xception follows the principles of the Inception architecture, at its core, which emphasises the use of multiple parallel convolutional pathways of varying kernel sizes to capture features at different spatial scales. However, Xception introduces a novel concept called "depthwise separable convolutions," which replaces the standard convolutional layers in Inception modules.
Depthwise separable convolutions decompose the standard convolution operation into two separate steps:

- depthwise convolutions: each channel of the input is convolved independently with a separate filter. This operation captures spatial relationships within each channel individually, reducing the computational cost compared to traditional convolutions;
- pointwise convolutions: a 1x1 convolution is applied across the depth of the feature maps to combine the information from different channels. This step allows for cross-channel feature interactions and dimensionality reduction, further enhancing the representational power of the network.

By replacing the standard convolutions with depthwise separable convolutions throughout the network, Xception achieves a significant reduction in the number of parameters and computational complexity while maintaining or even improving performance.
Xception represents a breakthrough in convolutional neural network design, leveraging depthwise separable convolutions to achieve a balance between performance and efficiency. Its innovative architecture has demonstrated state-of-the-art performance on various image classification benchmarks while being highly scalable and adaptable to different tasks and deployment scenarios.
For that reason its implementation was chosen between the existing CNN different architectures to compare its performance with the model of chapter 4.5.

5.1 First Model

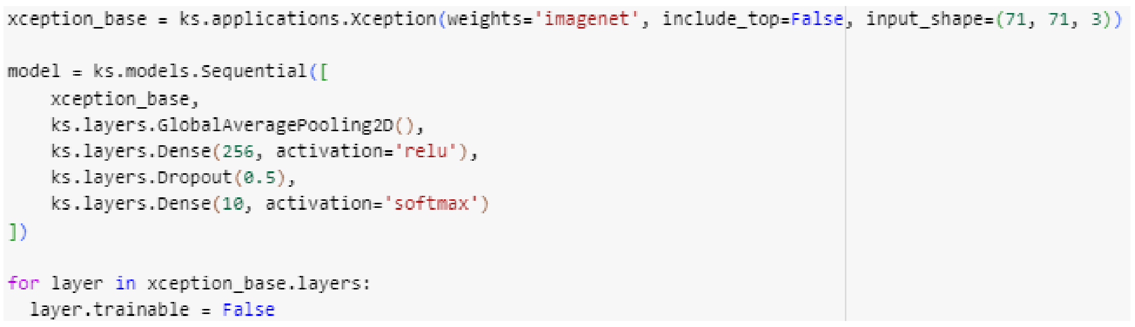
To the first Xception Model implemented, apart from the default parameters' values, there were considerate the following parameters:

Table 5.1.1 - Parameters considered for the first model.

Architecture	Xception architecture
GlobalAveragePooling2D*	After the Xception model layers
Dense layers	1 layer with 256 filters and all with Rectified Linear Unit activation function.
Dropout	A dropout layer of 0.5
Output layer	10 filters and Normalised exponential (softmax) activation function.
Optimizer	Adam Optimization Algorithm
Loss function	Sparse Categorical Cross Entropy

*Global Average Pooling 2D (GAP2D) is a technique commonly used in Convolutional Neural Networks (CNNs), especially in architectures like the Inception and MobileNet families. It serves several important purposes, such as dimensionality reduction, translation invariance, regularisation and scalability.

Also the resolution was set to 71x71 because 32x32 is low for Xception.
There were considered a total of 100 epochs to the training, and an early stopping using as monitor metric "validation loss” and “validation accuracy” and for both a patience equals to 10. Below one can see some of the relevant parts of the code implementing that model:

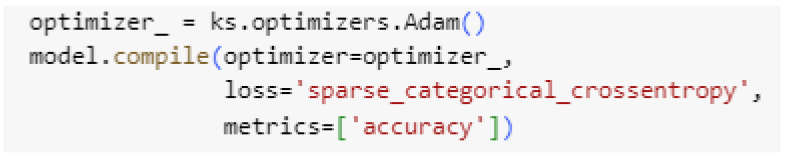


```
xception_base = ks.applications.Xception(weights='imagenet', include_top=False, input_shape=(71, 71, 3))

model = ks.models.Sequential([
    xception_base,
    ks.layers.GlobalAveragePooling2D(),
    ks.layers.Dense(256, activation='relu'),
    ks.layers.Dropout(0.5),
    ks.layers.Dense(10, activation='softmax')
])

for layer in xception_base.layers:
  layer.trainable = False
```

Fig 5.1.1 - Model definition (1st Xception model).



```
optimizer_ = ks.optimizers.Adam()
model.compile(optimizer=optimizer_,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Fig 5.1.2 - Definition of optimizer and loss function.

5.1.1 Results

Table 5.1.2 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
85.77%	72.32%	72.32%

Table 5.1.3 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.3866	1.0591

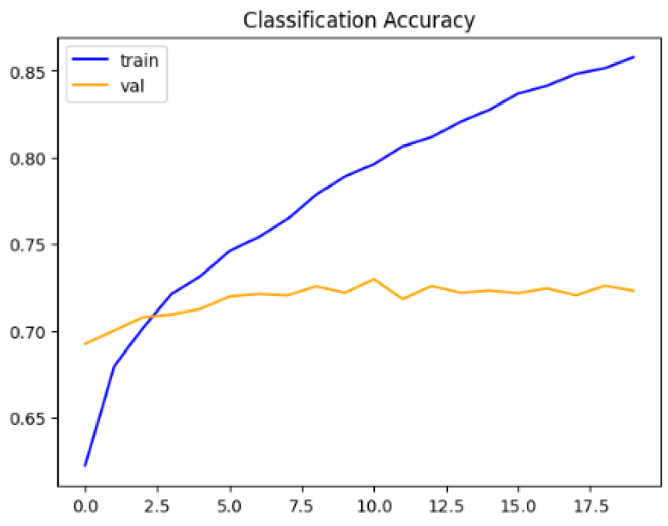
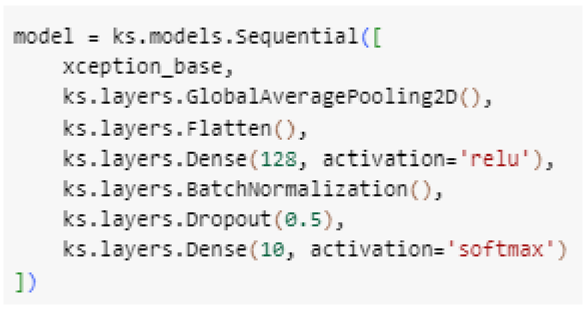


Fig 5.1.3 - Accuracy of train dataset (blue) and validation dataset (orange).

5.2 Second Model

To the second Xception Model implemented was based on the first, but instead of the dense layers and dropout used as before, it was inserted a flatten layer, a dense layers with 128 filters and a batch normalisation layer followed by a dropout of 0.5 (likewise the last layers of model 4.5). See the architecture in figure 5.2.1.



```
model = ks.models.Sequential([
    xception_base,
    ks.layers.GlobalAveragePooling2D(),
    ks.layers.Flatten(),
    ks.layers.Dense(128, activation='relu'),
    ks.layers.BatchNormalization(),
    ks.layers.Dropout(0.5),
    ks.layers.Dense(10, activation='softmax')
])
```

Fig 5.2.1 - Model definition (2nd Xception model).

5.2.1 Results

Table 5.2.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
79.36%	71.81%	71.81%

Table 5.2.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.5873	0.8511

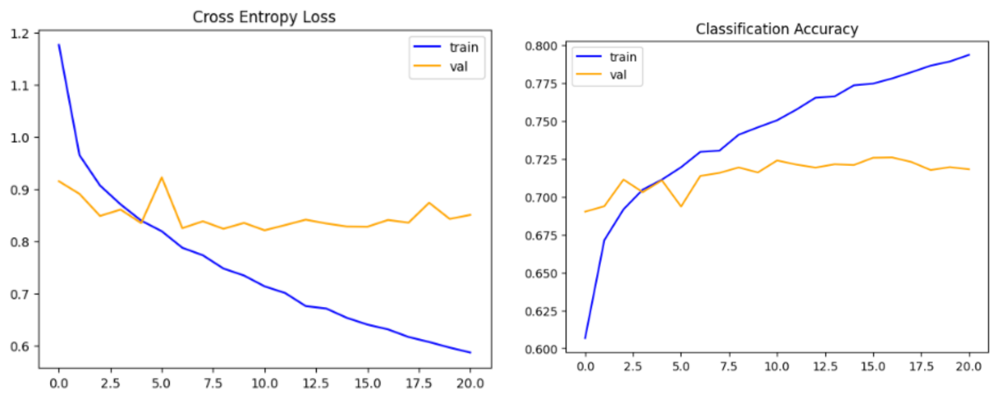


Fig 5.2.2 - Accuracy of train dataset (blue) and validation dataset (orange).

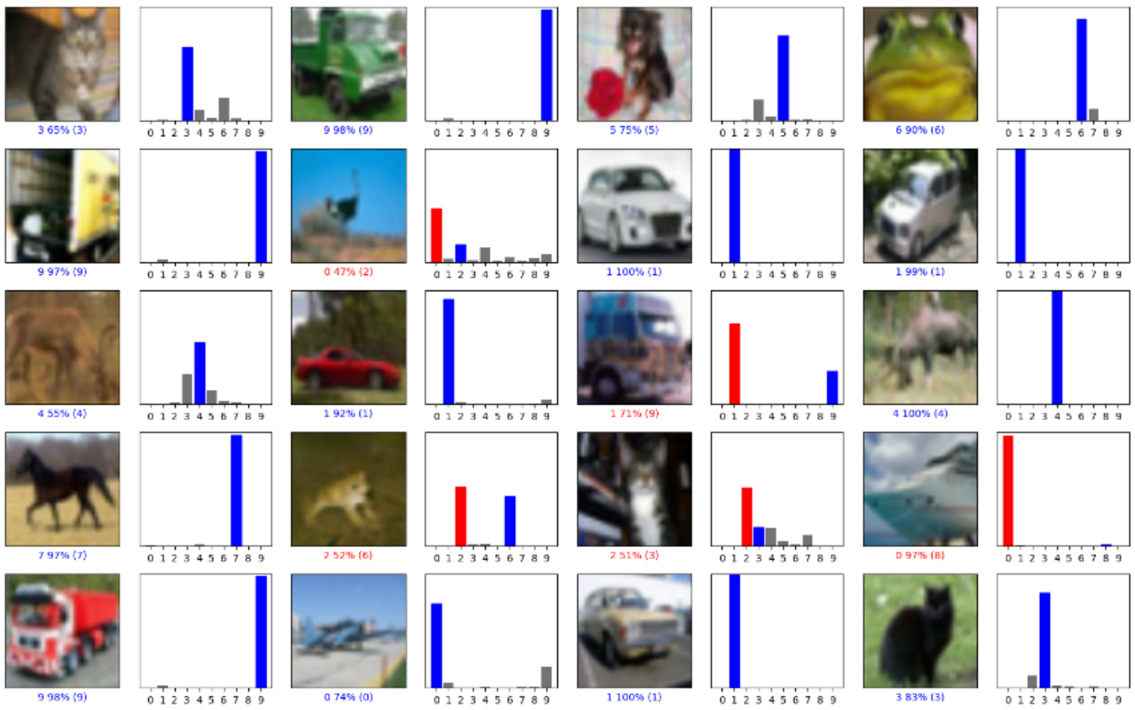


Fig 5.2.3 - Model Validation through images classification The blue colors on the graphs mean that the prediction was correct, contrarily to the red, which means that the model failed in correctly classifying the image.

6. Inception-V3 Architecture Models' description

Inception-v3 is a CNN architecture designed for image classification tasks, which represents a significant improvement over its predecessor, Inception-v1, with enhancements in both performance and computational efficiency.
One of the key features of Inception-v3 is its deep and intricate architecture, consisting of multiple layers of convolutional, pooling, and fully connected layers. It incorporates several innovative design principles to capture rich spatial hierarchies of features across different scales.
One of the most notable aspects of Inception-v3 is the use of "Inception modules," which are multi-branch convolutional blocks that perform **parallel convolutions** with different kernel sizes and pooling operations. This design allows the network to capture features at multiple resolutions and spatial extents, leading to better representation of complex patterns and structures in the input images.
Inception-v3 also introduces several optimization techniques to improve training stability and convergence speed. These include batch normalisation, which normalises the activations of each layer to stabilise training, and factorised 7x7 convolutions, which reduce the computational cost of convolutions by decomposing them into smaller operations.
Furthermore, Inception-v3 incorporates auxiliary classifiers at intermediate layers during training, which serve as regularisation mechanisms to prevent overfitting and provide additional supervision signals for training. This multi-task learning approach helps improve the generalisation performance of the model on unseen data.
So, like it was made to the Xception model, Inception-V3 architecture (Keras) was implemented to compare with all the previous models performed.

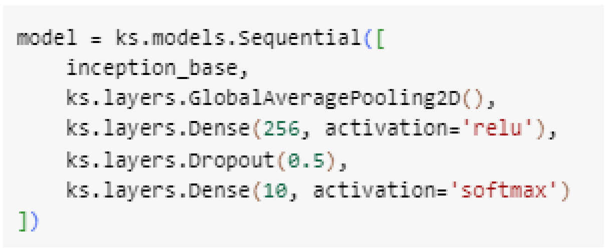
6.1 First Model

To the first Inception-V3 Model implemented, apart from the default parameters' values, there were considerate the following parameters:

Table 6.1.1 - Parameters considered for the first model.

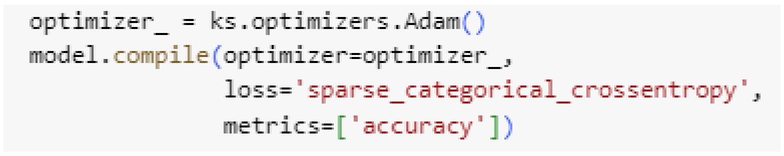
Architecture	Inception-V3 architecture
GlobalAveragePooling2D*	After the Inception model layers
Dense layers	1 layer with 256 filters and all with Rectified Linear Unit activation function.
Dropout	A dropout layer of 0.5
Output layer	10 filters and Normalised exponential (softmax) activation function.
Optimizer	Adam Optimization Algorithm
Loss function	Sparse Categorical Cross Entropy

Also the resolution was set to 71x71 because 32x32 is low for Xception.
There were considered a total of 100 epochs to the training, and an early stopping using as monitor metric "validation loss" and "validation accuracy" and for both a patience equals to 10. Below one can see some of the relevant parts of the code implementing that model:



```
model = ks.models.Sequential([
    inception_base,
    ks.layers.GlobalAveragePooling2D(),
    ks.layers.Dense(256, activation='relu'),
    ks.layers.Dropout(0.5),
    ks.layers.Dense(10, activation='softmax')
])
```

Fig 6.1.1 - Model definition (1st Xception model).



```
optimizer_ = ks.optimizers.Adam()
model.compile(optimizer=optimizer_,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Fig 6.1.2 - Definition of optimizer and loss function.

6.1.1 Results

Table 6.1.2 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
69.43%	62.20%	62.20%

Table 6.1.3 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.8645	1.1179

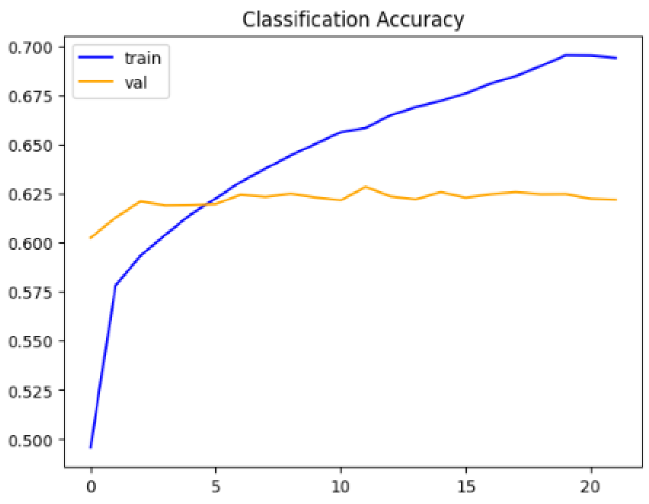
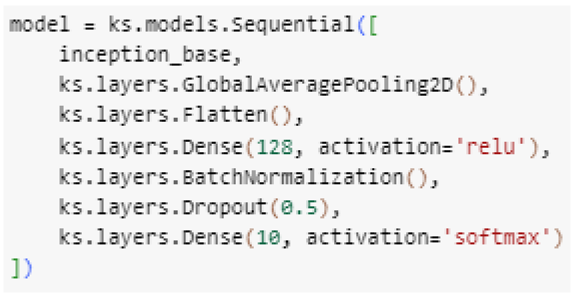


Fig 6.1.3 - Accuracy of train dataset (blue) and validation dataset (orange).

6.2 Second Model

To the second Inception-V3 Model implemented was based on the first, but instead of the dense layers and dropout used as before, it was inserted a flatten layer, a dense layers with 128 filters and a batch normalisation layer followed by a dropout of 0.5 (likewise the last layers of model 4.5). See the architecture in figure 5.2.1.



```
model = ks.models.Sequential([
    inception_base,
    ks.layers.GlobalAveragePooling2D(),
    ks.layers.Flatten(),
    ks.layers.Dense(128, activation='relu'),
    ks.layers.BatchNormalization(),
    ks.layers.Dropout(0.5),
    ks.layers.Dense(10, activation='softmax')
])
```

Fig 6.2.1 - Model definition (2nd Xception model).

6.2.1 Results

Table 6.2.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
68.90%	62.64%	62.64%

Table 6.2.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.8914	1.0992

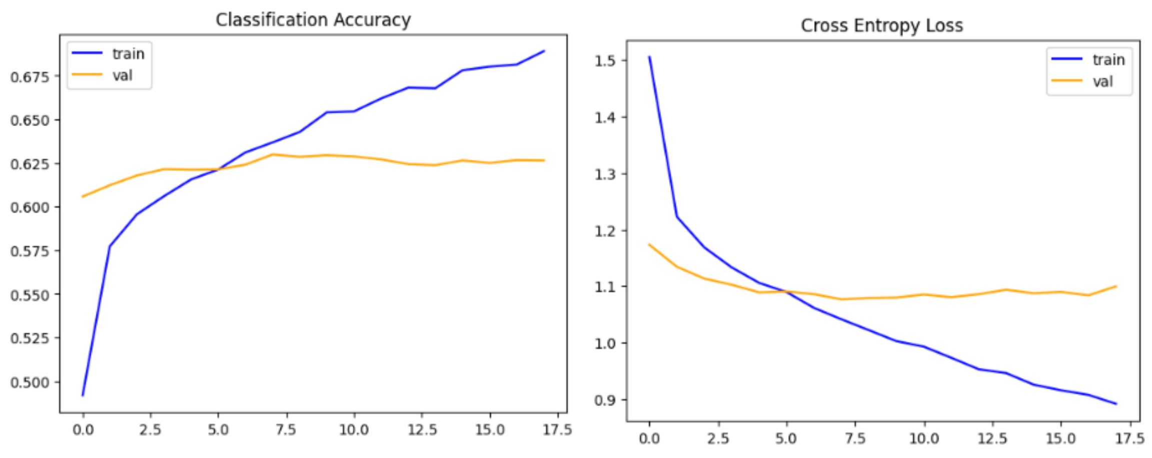


Fig 6.2.2 - Accuracy of train dataset (blue) and validation dataset (orange).

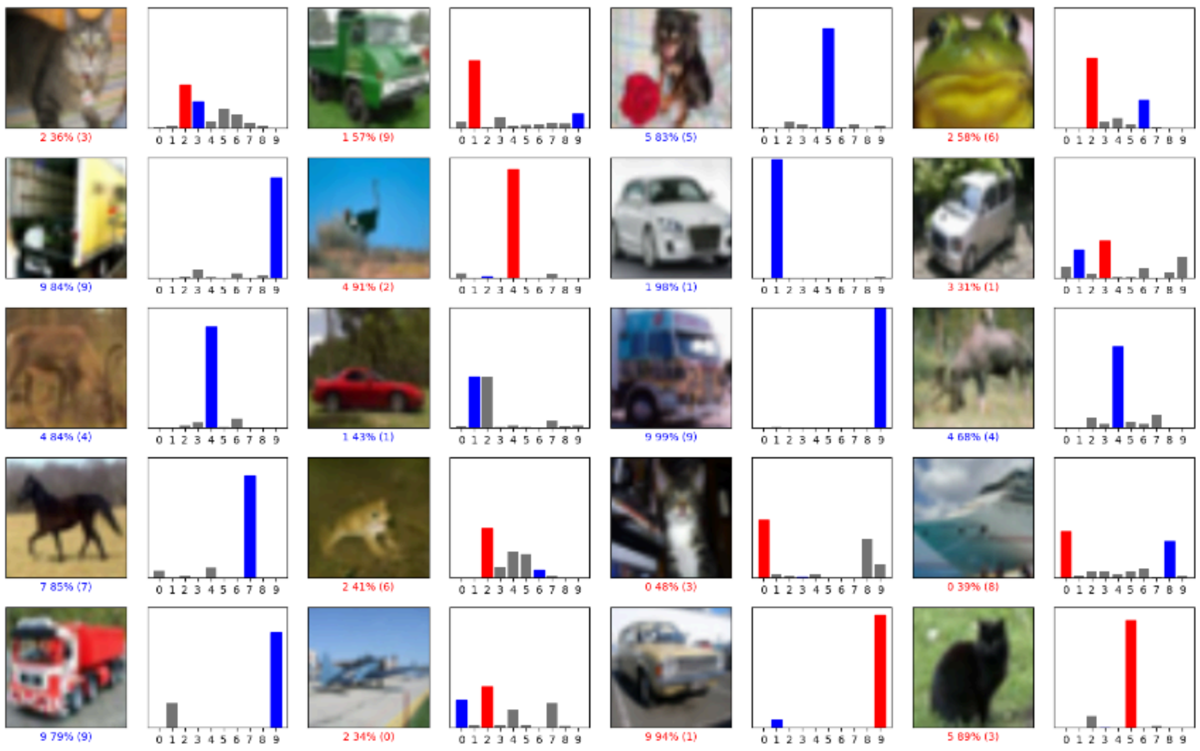


Fig 6.2.3 - Model Validation through images classification The blue colors on the graphs mean that the prediction was correct, contrarily to the red, which means that the model failed in correctly classifying the image.

7. Results and comparison

7.1 Accuracy

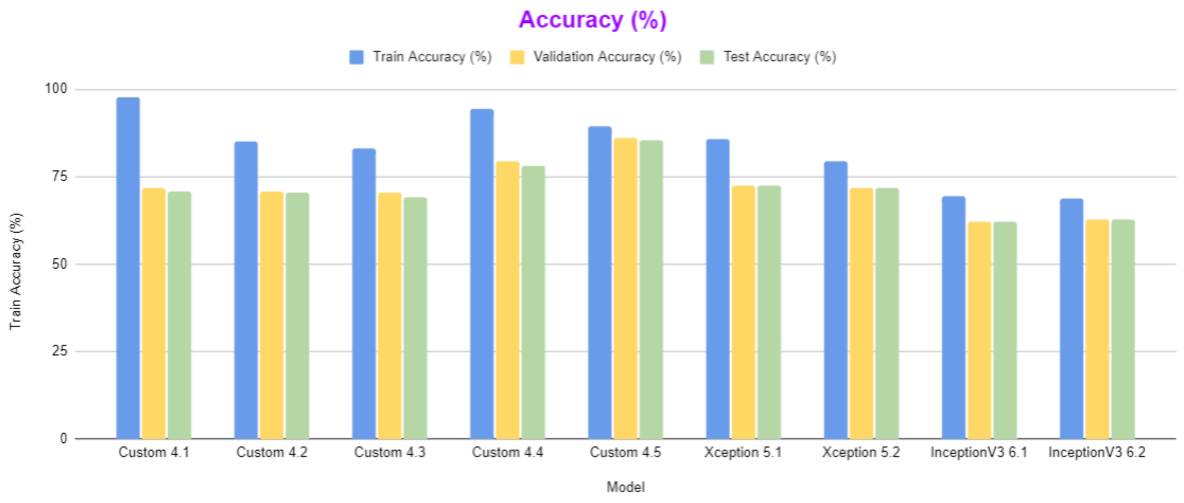


Fig 7.1.1 - Accuracies for all models performed.

By plotting side by side all the accuracies, one can see that, regarding the train accuracy, the 3 models that had the better performance are the custom sequential layers (4.1), the custom sequential layers (4.4) and the custom sequential layers (4.5). However, by looking for the difference between such values and the validation results, it's clear that the 4.1 has a high overfitting. The 4.4 still has, but it was reduced due to the dropout layers and the batch normalisation used on the architecture. On another hand, the 4.5 has a great performance and doesn't show to overfit the train set: 89% versus 85% accuracy for train and validation/test datasets is a very good result.

Regarding the Xception and Inception-V3 architectures, the results weren't so bad, overall. However not so good as for the models 4.4 and 4.5. It could have several explanations and those solutions need to be more explored and refined. Additionally the Xception had a better performance than the Inception and it could be due several reasons, such as, the fact the Xcpetion utilises depthwise separable convolutions, which decompose the standard convolution operation into depthwise and pointwise convolutions, which allows the model to capture spatial relationships more efficiently while reducing the number of parameters and computational complexities. Other reasons could be the model capacity, or the robustness to variations in data including changes in object position, scale, and orientation.

7.2 Losses

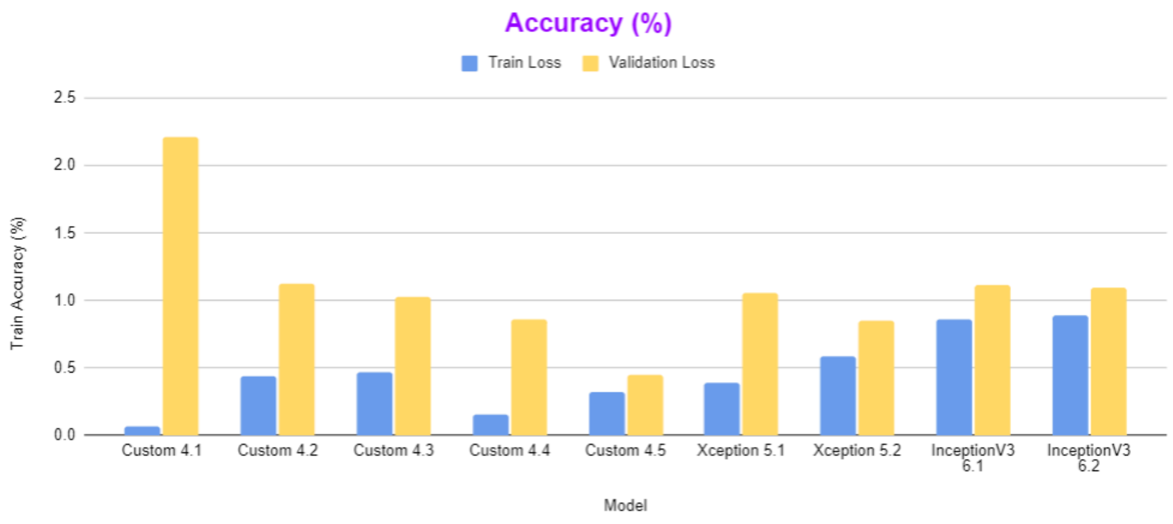


Fig 7.1.2 - Losses for all models performed.

Regarding the losses, the trend is similar to the accuracies, i.e, models with better performance tend to have lower losses and we can see that, in the model 4, from the first to fifth, the validation loss decreases continuously, which is a good indicator for the progression along the improvement of the model.
The validation losses among the Xception and Inception V3 models are similar, but the train' loss is clearly lower for the Xception Model.

7.3 Stability

In terms of stability, one can see by the figures 4.4.2 and 5.2.2, the Models Custom 4.4 and Xception 5.2 have some instability in the validation accuracy along the epochs. However it is not so relevant.
For the remaining cases, the evolution is very stable, having just some small variations along the training.

7.4 Validation through image classification

Regarding the validation through the 20 images classification, it was just considered for the models 4.5, 5.2 and 6.2.
By looking at the results, the model 4.5 (Figure 4.5.3) had a very good performance, in general, but it failed to correctly classify the boat and the ostriches, confusing it with an airplane. The case of the boat is quite understandable because it has very similarities with ariplanes (at least the current image of the boat). It also confused the cat with a dog.

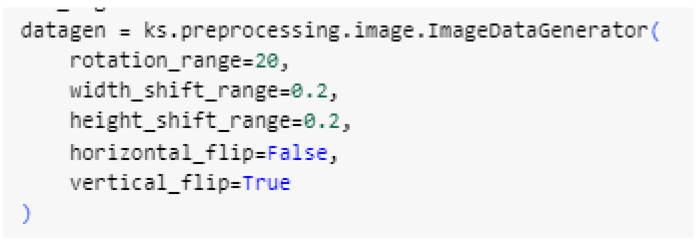
The Model 5.2 failed to classify 5 images, including the ostrich and the boat. However it classified the cat well.
At last, the model 6.2 failed to classify several images, and besides performing fine with the boat, by looking for all other classification, it could have a certain randomness on the result.
In the end, the model 4.5 had the better performance regarding the image classification.

8. Conclusions

After analysing all the models detailed before, it’s clear the model with better performance was the model 4.5. All the datasets were above 80% accuracy and it didn’t overfit the train dataset. Additionally in terms of stability, all the models had a reasonable to very good performance and the training time was less than 15 minutes for all of them.
The Xception and Inception-V3 models had acceptable results, but didn’t perform as good as the model 4.5. As further work it would be interesting to explore more those models to try better results, especially in the case of the Xception, which showed to have very potential.
Regarding the evaluation through the images classification, the models 4.5 and Xception 5.2 didn't have bad results, being interesting to try some improvements in both of them, but the model Inception-V3 6.2 had a very bad performance, giving the idea that it is classifying randomly the images.
At chapter 4.6.1 and 4.6.2 it was mentioned 2 studies to see the effect on changing activation functions of the output layer or the optimizer, and, besides not leading to relevant conclusions, it would be interesting to make more studies regarding those and other parameters.
At last, the appendix chapter shows the results obtained for the model 4.5 and Xception 6.1 considering the implementation of data augmentation technique (likewise shown in the appendixes), but the results were so bad that it was considered. It would be interesting to study such a technique, but it is something that requires time to explore.

9. Appendixes

9.1 Data augmentation for Xception 6.1 model



```
datagen = ks.preprocessing.image.ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=False,
    vertical_flip=True
)
```

Fig 9.1.1 - Data augmentation definition.

Table 9.1.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
9.94%	10.73%	10.73%

Table 9.1.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
1.1592	0.8757

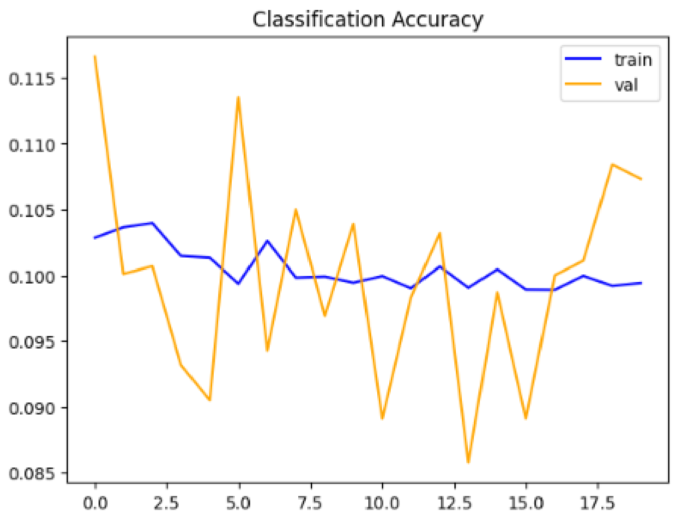


Fig 9.1.2 - Accuracy of train dataset (blue) and validation dataset (orange).

As one can see, the results are bad and the validation loss is very unstable.

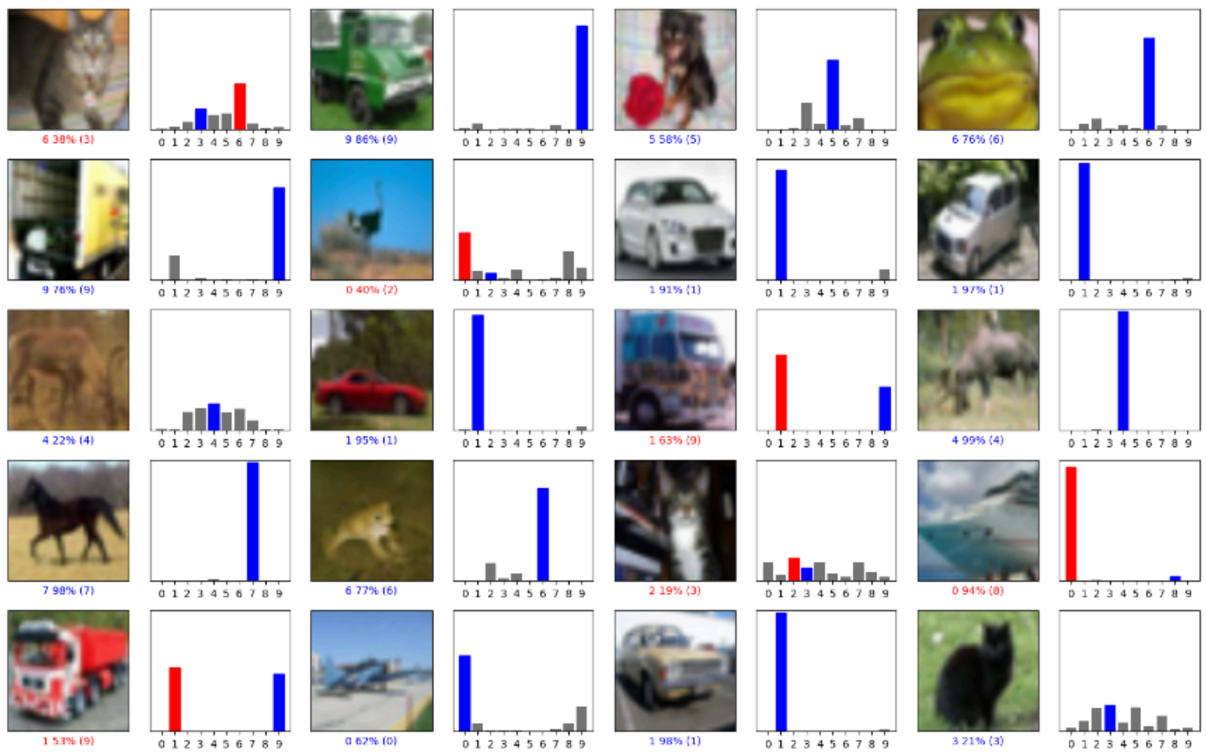
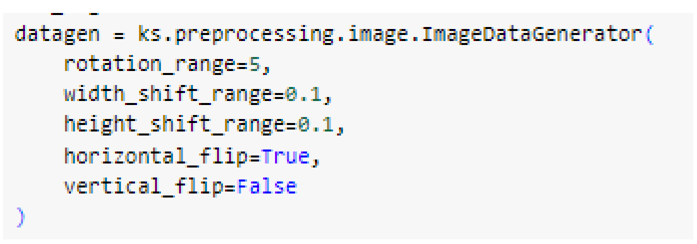


Fig 9.1.3 - Model Validation through images classification The blue colors on the graphs mean that the prediction was correct, contrarily to the red, which means that the model failed in correctly classifying the image.

9.2 Data augmentation for Custom 4.5 model



```
datagen = ks.preprocessing.image.ImageDataGenerator(
    rotation_range=5,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    vertical_flip=False
)
```

Fig 9.2.1 - Data augmentation definition.

Table 9.2.1 - Accuracies for train, validation and test datasets.

Train accuracy	Validation accuracy	Test accuracy
10.22%	8.85%	8.85%

Table 9.2.2 - Losses for train and Validation datasets.

Train Loss	Validation Loss
0.6240	0.5507

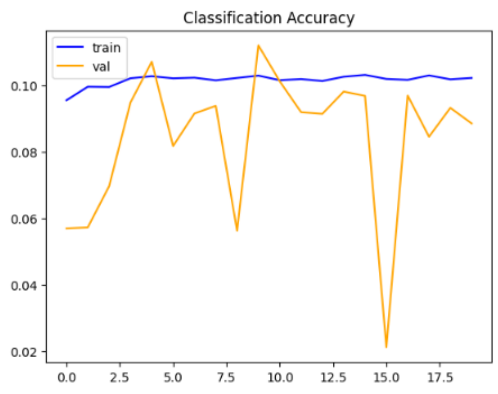


Fig 9.2.2 - Accuracy of train dataset (blue) and validation dataset (orange).

The behaviour is the same as for the last model with data augmentation.

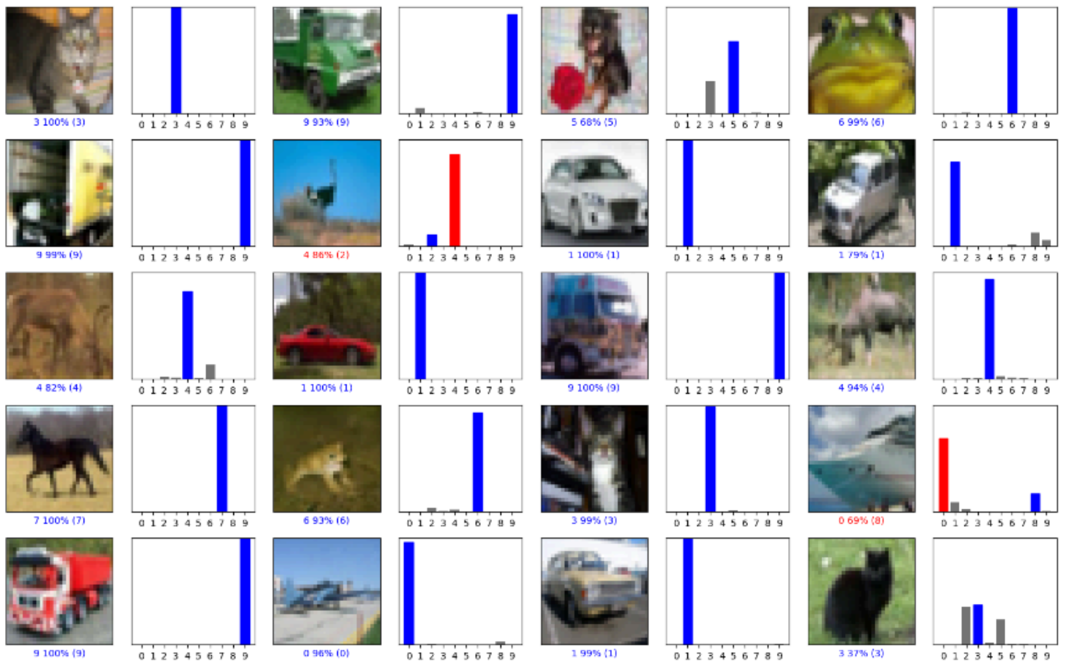


Fig 9.2.3 - Model Validation through images classification The blue colors on the graphs mean that the prediction was correct, contrarily to the red, which means that the model failed in correctly classifying the image.

10. Fonts

1. https://maelfabien.github.io/deeplearning/xception/
2. https://medium.com/@AnasBrital98/inception-v3-cnn-architecture-explained-691cfb7bba08
3. https://iq.opengenus.org/inception-v3-model-architecture/
4. Grus J. “Data Science from scratch”, 2nd edition, 2019