Escuela Técnica Superior de Ingenieros Informáticos | antes Facultad de Informática
Universidad Politécnica de Madrid

Deep Learning Project
Image Recognition Assignment

Master in Data Science
Deep Learning

Carlos Almodóvar Román
Enrique Martínez Martel
Juan Ramón Romero García

Contents

1. Introduction

Image recognition is a subset of computer vision and has emerged as an innovative technology with several applications across numerous industries and domains. It involves teaching computers to interpret and understand visual data, enabling them to identify objects, scenes, patterns, and even human faces in images or videos. This capability holds immense potential in revolutionizing various sectors, from healthcare and security to retail and automotive.

The significance of image recognition lies in its ability to augment human capabilities and automate tasks that were once reliant on manual intervention. Image recognition systems use sophisticated algorithms and machine learning methods to efficiently process and analyze large volumes of visual data, surpassing human abilities in speed. This advancement not only improves effectiveness and precision but also facilitates the creation of inventive solutions for intricate challenges.

In the healthcare sector, image recognition plays a crucial role in medical imaging, aiding clinicians in diagnosing diseases and guiding treatment decisions. From detecting tumors in medical scans to identifying abnormalities in X-rays and MRIs, image recognition technology enhances the accuracy and efficiency of diagnosis, ultimately improving patient outcomes and saving lives.

In the realm of security and surveillance, image recognition enables the monitoring and analysis of video feeds from surveillance cameras, identifying suspicious activities, and alerting security personnel in real-time. This enhances situational awareness and helps prevent security breaches, making public spaces safer and more secure.

Moreover, image recognition has revolutionized the retail industry, powering recommendation systems that personalize shopping experiences based on customer preferences and past behavior. By analyzing images of products and understanding their visual features, image recognition systems can suggest relevant items to users, increasing customer satisfaction and driving sales.

In the automotive sector, image recognition technology is integral to the development of autonomous vehicles, enabling them to perceive their surroundings and make informed decisions in real-time. From detecting pedestrians and obstacles to interpreting traffic signs and signals, image recognition systems enhance the safety and reliability of autonomous driving systems, paving the way for a future of safer and more efficient transportation.

Overall, image recognition marks a significant change in how we engage with technology and interpret our surroundings. As progress speeds up, the possibilities for applying image recognition become limitless, providing answers to critical societal issues. This report will thoroughly examine the principles, approaches, uses, and forthcoming developments in image recognition, investigating its game-changing influence across different sectors and fields.

2. Description of the Problem

In this assignment, the focus lies on tackling the image recognition problem using deep learning models. The task involves developing proficiency in utilizing Tensorflow/Keras [1] for training Neural Nets (NNs) and applying acquired knowledge to optimize the parameters and architecture of both feed-forward Neural Nets (ffNNs) and Convolutional Neural Nets (CNNs).

The dataset provided for this task is the xView [2] dataset, a large publicly available object detection dataset containing approximately one million objects across 60 categories.

One of the key challenges in this task is the severe class imbalance present in the dataset. This imbalance in class frequencies can significantly affect the overall accuracy of the models and must be addressed during the training process.

To address this problem, participants are required to design and train neural networks using both ffNNs and CNNs. They must decide on the number of layers and units in each layer of their NN, select optimization algorithms and parameters, and monitor the evolution of these parameters using a validation subset to decide when to stop training. Additionally, participants are encouraged to explore regularization methods such as dropout and data augmentation to enhance model performance.

Furthermore, participants are tasked with comparing their architecture and results with other popular architectures such as GoogLeNet [3], VGG [4], and ResNet [5]. They are also required to train some of these popular architectures from scratch and compare the results with pre-trained versions using transfer learning.

In summary, the problem involves designing and training neural networks to tackle the image recognition task using the xView dataset. The goals are to optimize the models, address class imbalance issues, and compare results with popular architectures to gain insights into the performance and effectiveness of different approaches. The ultimate goal is to develop robust image recognition models capable of accurately identifying objects in images across various categories.

3. Description of the Data Sets

The xView dataset is essential for the image recognition task in this assignment. This dataset is publicly available and curated for the purpose of advancing research in computer vision and machine learning. The images in the xView dataset are manually annotated and sourced from various scenes around the world, captured using the high-resolution WorldView-3 satellite with a ground sample distance of 0.3 meters. This high-resolution imagery ensures detailed and accurate representations of objects and scenes, making it suitable for training and evaluating image recognition models.

xView dataset has been divided into training and testing subsets, the training subset contains a total of 21377 objects, while the testing subset contains 2635 objects. These objects have been extracted from a pool of 846 annotated images in the dataset, 761 and 85 images for training and testing respectively. To facilitate the image recognition task, the dataset has been preprocessed to focus on the 12 most represented classes.

Category	Count	Frequency
Cargo plane	635	2.97%
Helicopter	70	0.32%
Small Car	4290	20.06%
Bus	2155	10.08%
Truck	2746	12.84%
Motorboat	1069	5.00%
Fishing Vessel	706	3.30%
Dump Truck	1236	5.78%
Excavator	789	3.69%
Building	4689	21.93%
Storage Tank	1469	6.87%
Shipping Container	1523	7.12%

Table 1: Data set distribution

Each image in the dataset is annotated with bounding boxes, highlighting the location and extent of objects within the image. These annotations provide ground truth labels for training and evaluating image recognition models, enabling the development of accurate and robust algorithms for object detection and classification. In addition to the annotated images, the dataset also includes metadata such as class frequencies, providing insights into the distribution of objects across different categories. This information is crucial for understanding the class imbalance present in the dataset and devising strategies to mitigate its impact on model performance.

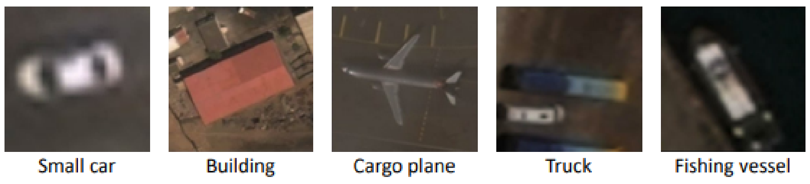


Figure 1: Sample objects from the dataset

4. Process Followed and Intermediate Results

Firstly, we tried to create a neural network with no convolutional layers. All of the layers had to be Dense layers and the computational cost was too high. Times to be executed almost seemed like half day computing and training and testing.

This constraint also gave us the constraint of not using MaxPooling2D and instead Flatten layer was used.

A neural network in Keras is created the following way: First, create a model of the class Sequential, which indicates that the layers will be placed in a sequential way. After this, the specification of the input shape of the data. Now specify with the “add()” method each type of layer with its parameters, activation function, number of neurons, etc. WIth these, for the ending part, the most common way to end the neural network is with a “softmax” layer, which will classificate the number of different labels that the class variable will have.

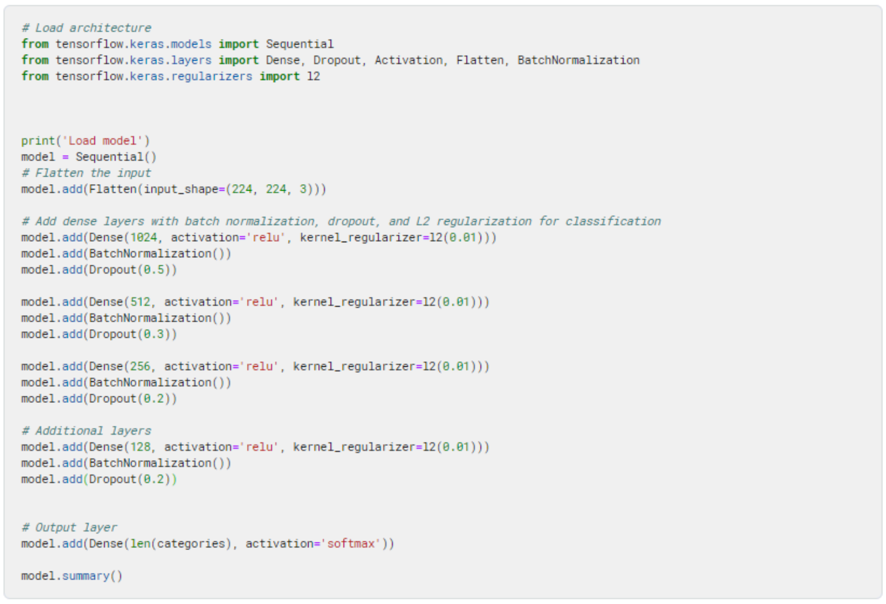
Later on, some techniques to improve and increase the performance of the neural layer like callbacks, a callback is a mechanism that allows you to customize the behavior of the training process at various stages, such as at the start or end of an epoch, before or after a batch, or when a model is saved or restored. Callbacks are useful for tasks like logging training metrics, saving model checkpoints, adjusting learning rates dynamically, or stopping early to prevent overfitting. They are implemented as functions or objects that are passed to the training process and called at specific points during training. Callbacks enable flexibility and control over the training process, allowing you to monitor and adapt the training based on various conditions and requirements. Another technique would be the early stopping, early stopping is a technique in neural network training where the training process is halted if the model's performance on a validation dataset fails to improve for a certain number of epochs. This helps prevent overfitting and ensures the model generalizes well to unseen data. It's implemented using callbacks, monitoring a chosen metric like validation loss or accuracy, and stopping training when improvement stagnates.

Having all of these, after that, an optimizer is created, in this study only “Adam” and “Nadam” will be used to compute different neural networks. These optimizers like Adam and Nadam are essential in neural network training, updating model parameters to minimize loss. In this optimizer, the loss function and the accuracy metric will be passed through parameters. The loss function this time will be “categorial_crossentropy” and the metric will be “accuracy”.

With this, the part of the model to be trained comes up. Is in this part where a specific subset of data has to be included in the parameters. For this training part, the model will have to divide the whole dataset into two different parts. The train data, called xview-recognition-train, will be divided into train sets and validation sets. This is because the model will be evaluated several times, called epochs. In each epoch, the model will learn and the accuracy and the loss function will grow and diminish respectively in order to create a better model for the testing phase.

4.1 Feed-Forward Neural Network

After testing with different architectures for the ffNN, the best one we obtained can be seen in the figure below.



```python
# Load architecture
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from tensorflow.keras.regularizers import l2



print('Load model')
model = Sequential()
# Flatten the input
model.add(Flatten(input_shape=(224, 224, 3)))

# Add dense layers with batch normalization, dropout, and L2 regularization for classification
model.add(Dense(1024, activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(512, activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.3))

model.add(Dense(256, activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.2))

# Additional layers
model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.2))


# Output layer
model.add(Dense(len(categories), activation='softmax'))

model.summary()
```

Figure 2: FFNN architecture

In this architecture we have several layers like the one at the input, the flatten layer for the input data and after this, several blocks of Dense, BatchNormalization and Dropout. Batch normalization normalizes the activations of each layer during training, stabilizing and accelerating the process by reducing internal covariate shift. Dropout randomly deactivates neurons during training, preventing overfitting by forcing the network to learn robust representations. These techniques improve performance and generalization in deep neural networks. For this network, we obtained the following results.

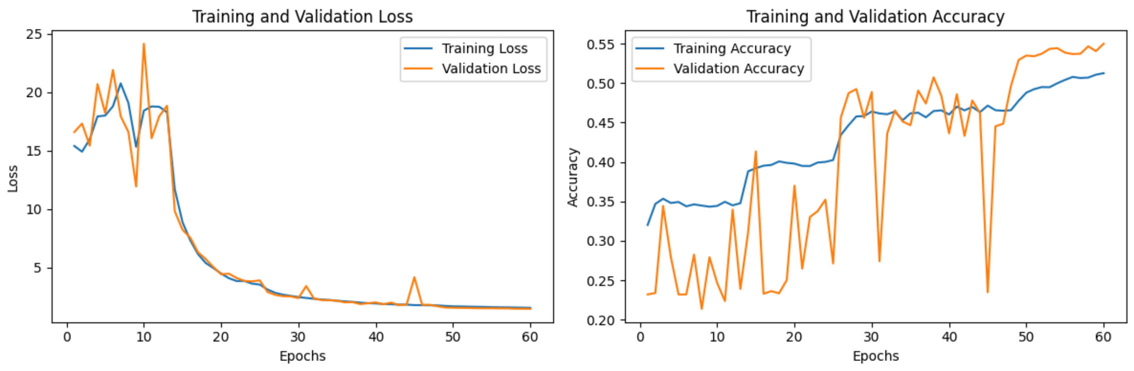
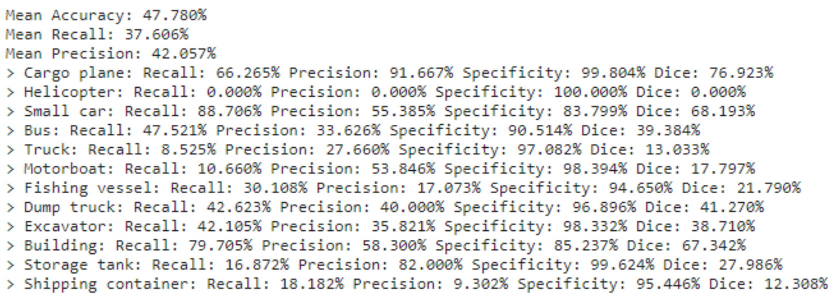


Figure 3: FFNN loss and accuracy

The model tries its best with this dataset with the constraint of not having convolutional layers and almost reached the 50% of the accuracy of the evaluation metric. The hyperparameter epoch takes an important part in the training part. That is, when surpassing a certain number of epochs, the loss function is not getting any lower, however when looking at the accuracy, in some cases the validation subset is not correctly classified and the results obtained may get worse in future epochs due to that. On the other hand, when looking at the training accuracy, which is the main indicator that the model is learning correctly, the model keeps getting better.

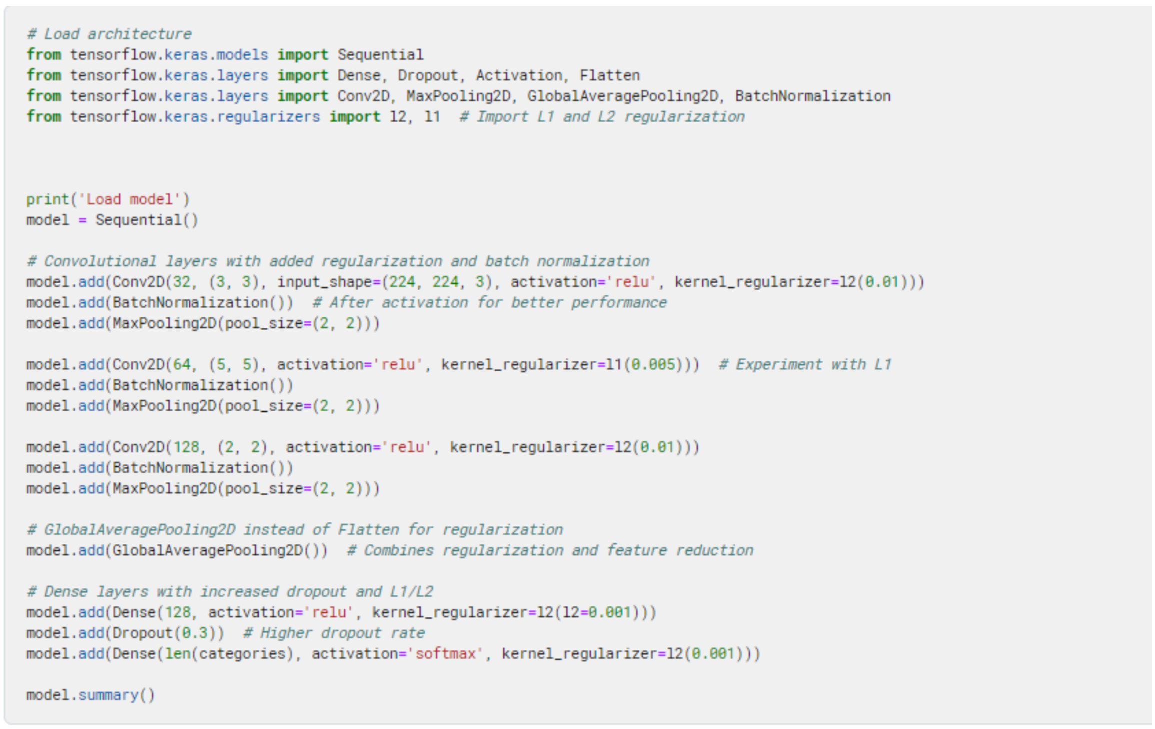


```
Mean Accuracy: 47.780%
Mean Recall: 37.606%
Mean Precision: 42.057%
> Cargo plane: Recall: 66.265% Precision: 91.667% Specificity: 99.804% Dice: 76.923%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Small car: Recall: 88.706% Precision: 55.385% Specificity: 83.799% Dice: 68.193%
> Bus: Recall: 47.521% Precision: 33.626% Specificity: 90.514% Dice: 39.384%
> Truck: Recall: 8.525% Precision: 27.660% Specificity: 97.082% Dice: 13.033%
> Motorboat: Recall: 10.660% Precision: 53.846% Specificity: 98.394% Dice: 17.797%
> Fishing vessel: Recall: 30.108% Precision: 17.073% Specificity: 94.650% Dice: 21.790%
> Dump truck: Recall: 42.623% Precision: 40.000% Specificity: 96.896% Dice: 41.270%
> Excavator: Recall: 42.105% Precision: 35.821% Specificity: 98.332% Dice: 38.710%
> Building: Recall: 79.705% Precision: 58.300% Specificity: 85.237% Dice: 67.342%
> Storage tank: Recall: 16.872% Precision: 82.000% Specificity: 99.624% Dice: 27.986%
> Shipping container: Recall: 18.182% Precision: 9.302% Specificity: 95.446% Dice: 12.308%
```

Figure 4: FFNN testing report

4.2 Convolutional Neural Network

This time, with the convolutional layers, the model was designed the following way: three different main blocks with an incremental number of neurons to learn more abstract and complex features built upon simpler features learned in earlier layers. Thanks to the convolutional layer, the MaxPooling layer is enabled to be used because it reduces the spatial dimensions of the feature maps while preserving the most important features, enabling the network to capture hierarchical representations of the input data more effectively.

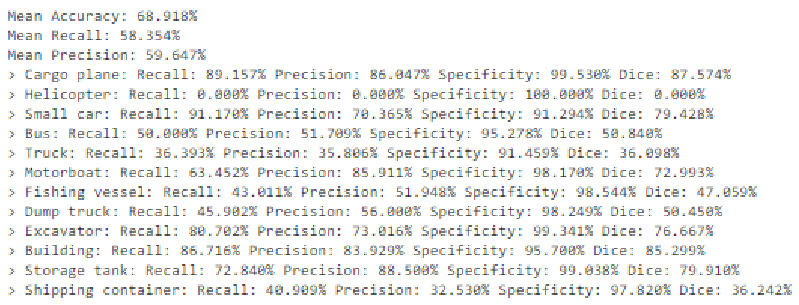


```python
# Load architecture
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, BatchNormalization
from tensorflow.keras.regularizers import l2, l1  # Import L1 and L2 regularization



print('Load model')
model = Sequential()

# Convolutional layers with added regularization and batch normalization
model.add(Conv2D(32, (3, 3), input_shape=(224, 224, 3), activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())  # After activation for better performance
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (5, 5), activation='relu', kernel_regularizer=l1(0.005)))  # Experiment with L1
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, (2, 2), activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

# GlobalAveragePooling2D instead of Flatten for regularization
model.add(GlobalAveragePooling2D())  # Combines regularization and feature reduction

# Dense layers with increased dropout and L1/L2
model.add(Dense(128, activation='relu', kernel_regularizer=l2(l2=0.001)))
model.add(Dropout(0.3))  # Higher dropout rate
model.add(Dense(len(categories), activation='softmax', kernel_regularizer=l2(0.001)))

model.summary()
```

Figure 5: CNN architecture

At the end of the code we have the GlobalAveragePooling2D which works better than fully connected layers because of its parameter reduction by computing the average of each feature map. Also, the same regularization techniques as Dropout, Batch Normalization and L1/L2 regularization are used to reduce the overfitting of the model.

With this scenario we had so much better results compared to the FFNN. However, even though the convolutional layers reduce dimensionality by gathering relevant features, the computational costs still remain very high and the time in creating the model is very large.



```
Mean Accuracy: 68.918%
Mean Recall: 58.354%
Mean Precision: 59.647%
> Cargo plane: Recall: 89.157% Precision: 86.047% Specificity: 99.530% Dice: 87.574%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Small car: Recall: 91.170% Precision: 70.365% Specificity: 91.294% Dice: 79.428%
> Bus: Recall: 50.000% Precision: 51.709% Specificity: 95.278% Dice: 50.840%
> Truck: Recall: 36.393% Precision: 35.806% Specificity: 91.459% Dice: 36.098%
> Motorboat: Recall: 63.452% Precision: 85.911% Specificity: 98.170% Dice: 72.993%
> Fishing vessel: Recall: 43.011% Precision: 51.948% Specificity: 98.544% Dice: 47.059%
> Dump truck: Recall: 45.902% Precision: 56.000% Specificity: 98.249% Dice: 50.450%
> Excavator: Recall: 80.702% Precision: 73.016% Specificity: 99.341% Dice: 76.667%
> Building: Recall: 86.716% Precision: 83.929% Specificity: 95.700% Dice: 85.299%
> Storage tank: Recall: 72.840% Precision: 88.500% Specificity: 99.038% Dice: 79.910%
> Shipping container: Recall: 40.909% Precision: 32.530% Specificity: 97.820% Dice: 36.242%
```

Figure 6: CNN Accuracy, recall and precision

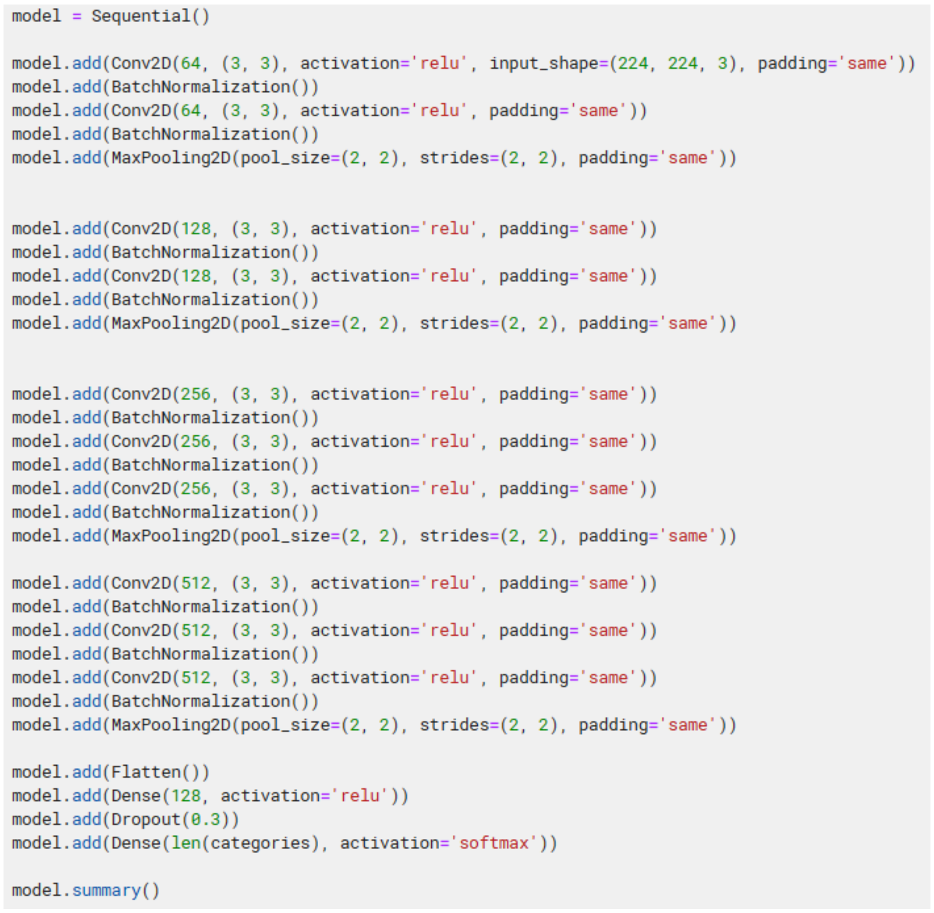
4.3 Popular architectures

4.3.1 VGG16 from scratch

VGG16 is a 16-layer convolutional neural network architecture designed for image recognition. It achieves high accuracy through a simple and uniform structure of small filters stacked in multiple layers, making it easy to understand and implement. Despite its simplicity, VGG16 achieved impressive performance on the ImageNet [6] dataset and remains a popular choice for transfer learning in various computer vision tasks.

After trying all of these “home-made” neural networks by changing parameters and attempting to get better accuracy, we opted to try this popular architecture, VGG16 but first training it from scratch.

As we can see in the following code, VGG16 has a typical convolutional neural network structure. The first block extracts features through 2 convolutional layers with 64 3x3 filters and ReLu activation, followed by batch normalization and max pooling. Subsequent blocks (2, 3, and 4) progressively increase feature complexity by doubling the number of filters in each convolutional layer (128, 256, and 512 respectively) and adding an extra convolutional layer per block. Finally, the extracted features are flattened and fed into dense layers for classification. A dropout layer helps prevent overfitting, and the final output layer uses a softmax activation to predict the class probabilities.



```
model = Sequential()

model.add(Conv2D(64, (3, 3), activation='relu', input_shape=(224, 224, 3), padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))


model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))


model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))

model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(len(categories), activation='softmax'))

model.summary()
```

Figure 7: VGG16 NN architecture

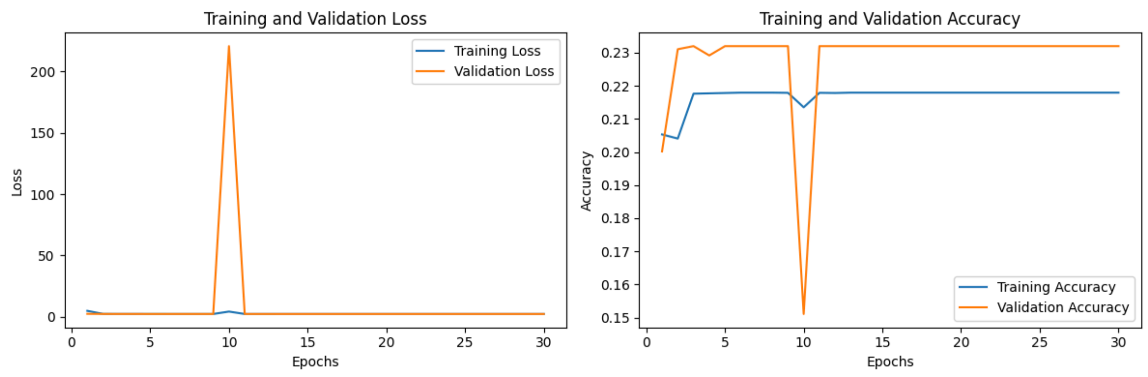
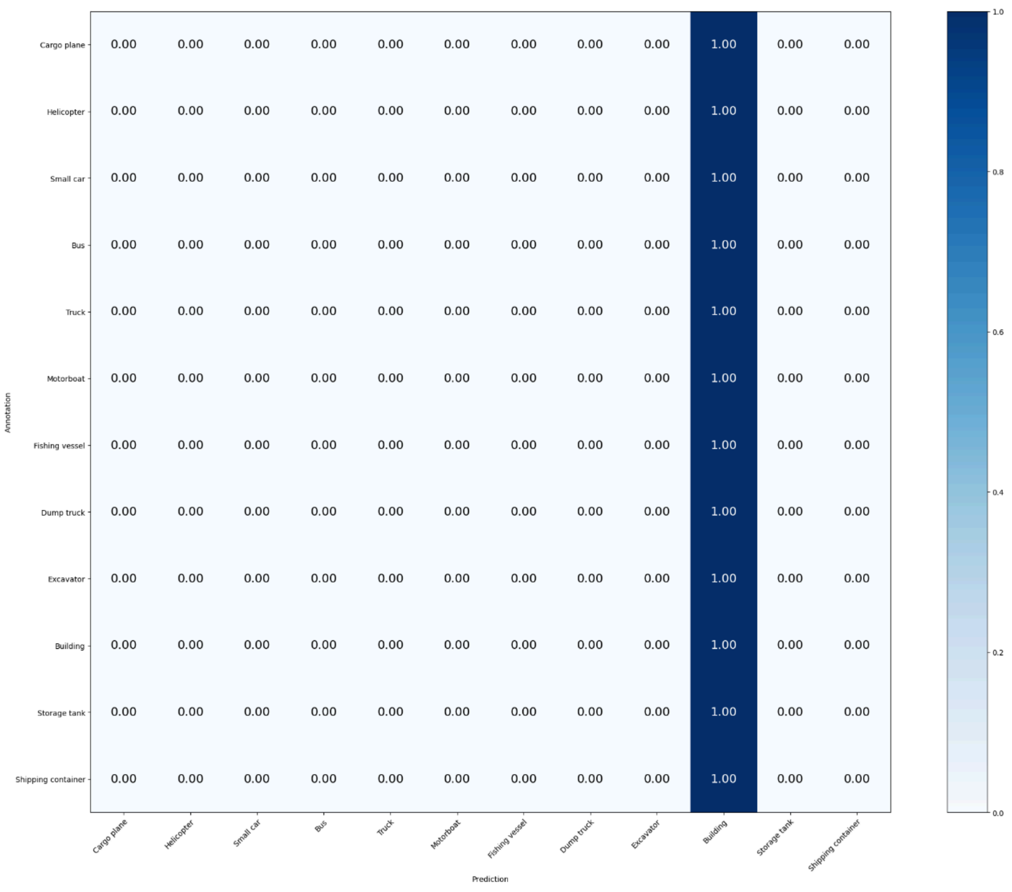


Figure 8: VGG16 NN loss function and accuracy

The results obtained were not good. Therefore, we will try to apply transfer-learning and use a pretrained model for this architecture to compare performances.

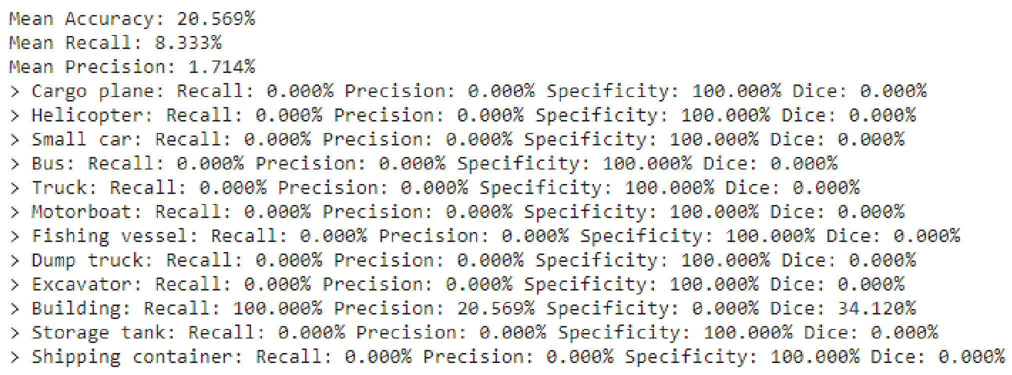
In the confusion matrix figure we can see how our model is performing. In this case, our model only classifies the instances as ‘buildings’.



Annotation \ Prediction	Cargo plane	Helicopter	Small car	Bus	Truck	Motorboat	Fishing vessel	Dump truck	Excavator	Building	Storage tank	Shipping container
Cargo plane	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Helicopter	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Small car	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Bus	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Truck	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Motorboat	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Fishing vessel	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Dump truck	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Excavator	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Building	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Storage tank	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Shipping container	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00

Figure 9: VGG16 NN confusion matrix

Finally, we can see that the results from the VGG16 architecture from scratch are bad, the accuracy is just 20%. So, taking into account the confusion matrix and the results it is very obvious that the configuration of the architecture and parameters are not the correct one.



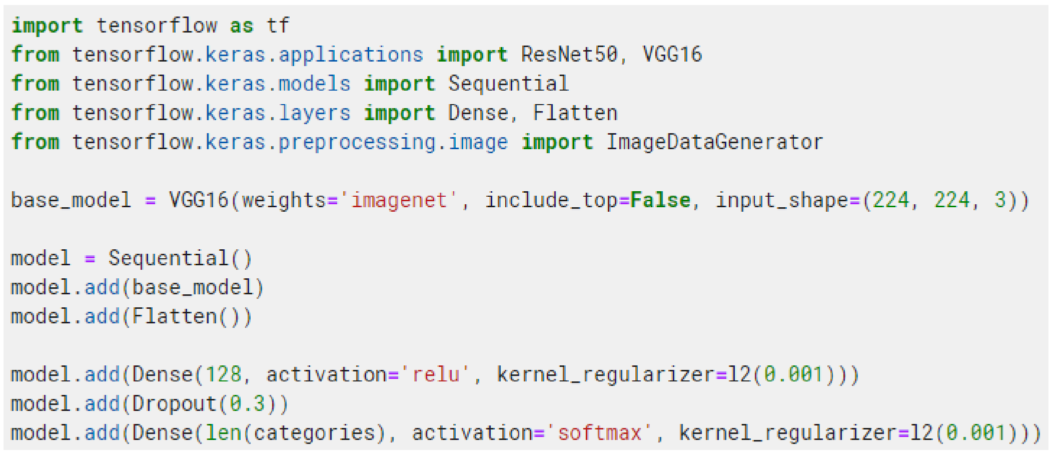
```
Mean Accuracy: 20.569%
Mean Recall: 8.333%
Mean Precision: 1.714%
> Cargo plane: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Small car: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Bus: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Truck: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Motorboat: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Fishing vessel: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Dump truck: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Excavator: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Building: Recall: 100.000% Precision: 20.569% Specificity: 0.000% Dice: 34.120%
> Storage tank: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Shipping container: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
```

Figure 10: VGG16 NN Testing Results

4.3.2 VGG16 with Transfer Learning

In this case we will be using a VGG16 architecture with transfer learning. We expect that it will help us to achieve better results.

To obtain the VGG16 pretrained model we used Keras and Tensorflow. Afterwards, we flatten and added a fully connected layer with 128 neurons, a ReLU activation function and a L2 regularization. L2 regularization is a technique used to prevent overfitting by penalizing models with large weights. We added dropout and, at the end, the fully connected layer with the softmax activation function that is responsible for classifying the input image into one of the categories.



```python
import tensorflow as tf
from tensorflow.keras.applications import ResNet50, VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.preprocessing.image import ImageDataGenerator

base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

model = Sequential()
model.add(base_model)
model.add(Flatten())

model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.3))
model.add(Dense(len(categories), activation='softmax', kernel_regularizer=l2(0.001)))
```

Figure 11: VGG16 NN with Transfer Learning Architecture

Analyzing the confusion matrix reveals the model's strengths and weaknesses in vehicle classification. While excelling at identifying unique vehicles like helicopters and excavators, it struggles with similar categories like sports cars and tracks. Furthermore, the model seems confused by the class "showing," often mistaking it for various vehicles. This suggests a need for improved differentiation between visually similar classes or a reevaluation of the "showing" category itself.

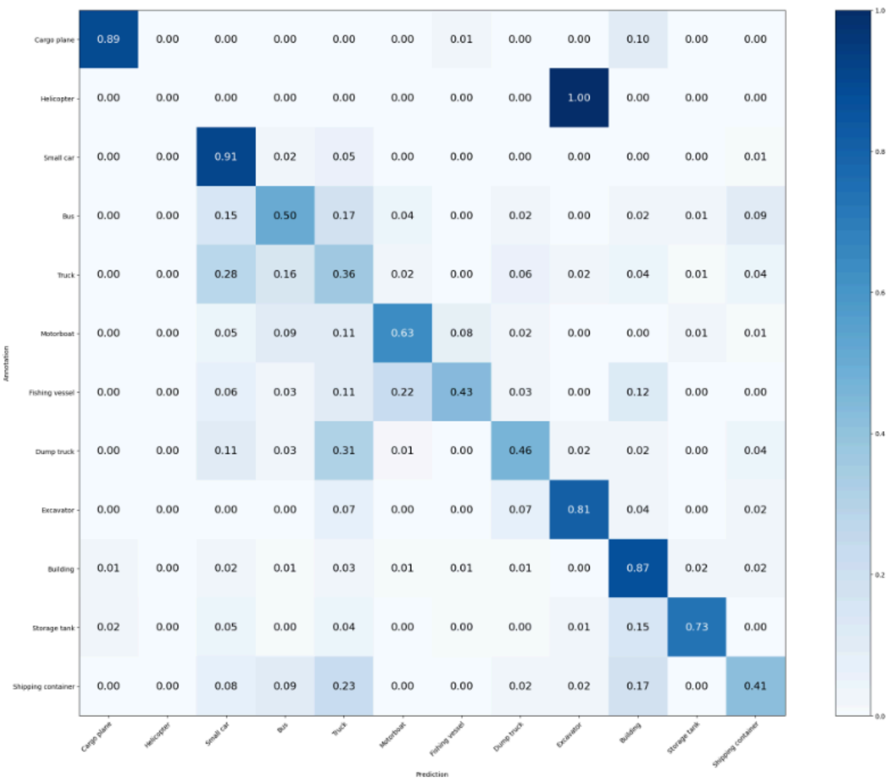
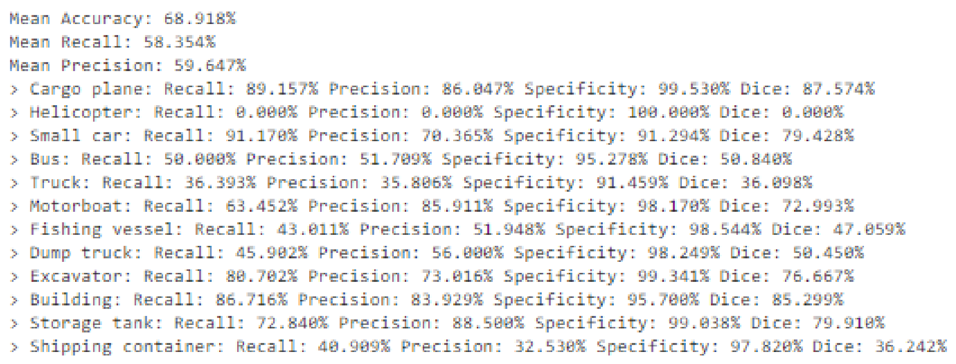


Figure 12: VGG16 NN with Transfer Learning Confusion Matrix

Finally, the results obtained in the testing were: a mean accuracy of 68%, a mean recall of 58% and a mean precision of 59%. If we check in more detail, we can see how categories like ‘cargo plane’ have a good accuracy of 89% while other categories such as ‘helicopter’ have a really bad result (0% of accuracy). The model's performance varies for other categories as well, with accuracy ranging from 40% to 70% for small cars and buses, where it often confuses between the two (19% misclassification rate). Overall, the model performs well on prominent object categories but needs improvement in differentiating rare or visually similar ones.



```
Mean Accuracy: 68.918%
Mean Recall: 58.354%
Mean Precision: 59.647%
> Cargo plane: Recall: 89.157% Precision: 86.047% Specificity: 99.530% Dice: 87.574%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Small car: Recall: 91.170% Precision: 70.365% Specificity: 91.294% Dice: 79.428%
> Bus: Recall: 50.000% Precision: 51.709% Specificity: 95.278% Dice: 50.840%
> Truck: Recall: 36.393% Precision: 35.806% Specificity: 91.459% Dice: 36.098%
> Motorboat: Recall: 63.452% Precision: 85.911% Specificity: 98.170% Dice: 72.993%
> Fishing vessel: Recall: 43.011% Precision: 51.948% Specificity: 98.544% Dice: 47.059%
> Dump truck: Recall: 45.902% Precision: 56.000% Specificity: 98.249% Dice: 50.450%
> Excavator: Recall: 80.702% Precision: 73.016% Specificity: 99.341% Dice: 76.667%
> Building: Recall: 86.716% Precision: 83.929% Specificity: 95.700% Dice: 85.299%
> Storage tank: Recall: 72.840% Precision: 88.500% Specificity: 99.038% Dice: 79.910%
> Shipping container: Recall: 40.909% Precision: 32.530% Specificity: 97.820% Dice: 36.242%
```

Figure 13: VGG16 NN with Transfer Learning Testing Results

As we say, the performance of the transfer-learning structure improves drastically. The differences are huge, having a pre-trained model, with actual knowledge trained, make these neural networks better at image recognition.

4.3.3 ResNet with Transfer Learning

Although the performance of the VGG16 had a poor performance in the first case (from scratch) and improved in the second case (with transfer learning) , we wanted to train a new popular architecture like the ResNet50 to compare performances with these structures. Therefore, for this approach we decided to go directly with transfer learning instead of building and training a ResNet50 from scratch.

What makes ResNet special is its ability to dive deep into understanding images by stacking layers of 'residual blocks'. These blocks help the network learn more efficiently by focusing on the difference between what it expects to see and what it actually sees, rather than starting from scratch each time. This clever technique makes it easier for the network to learn and remember complex patterns in images.

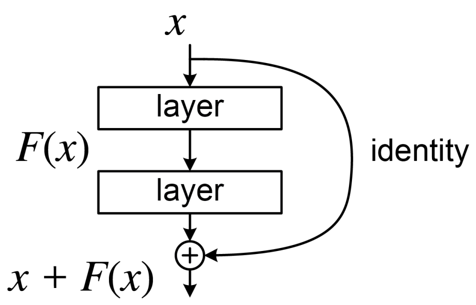
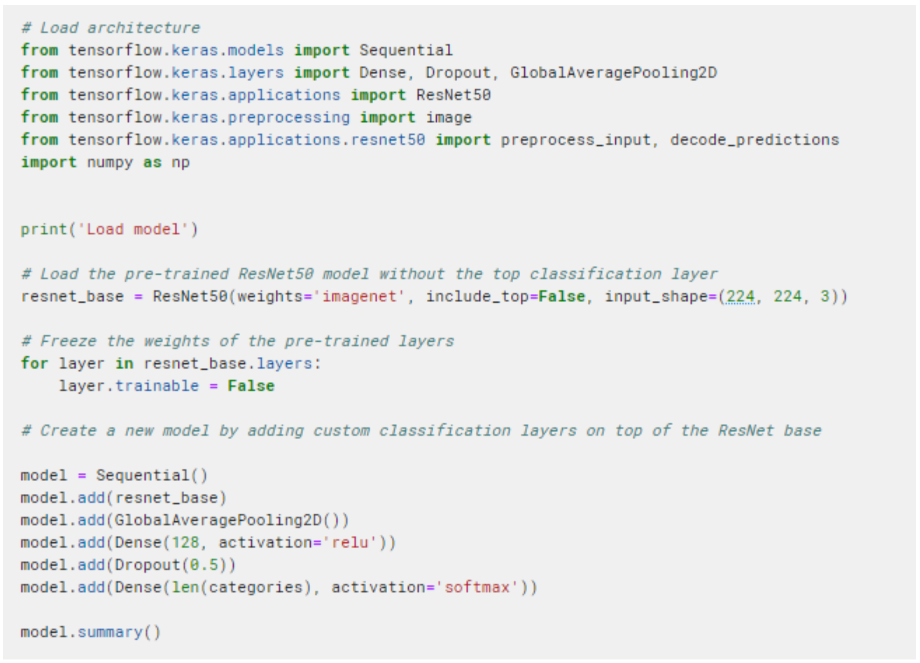


Figure 14: Residual Block for ResNet

Each ResNet block contains layers of operations like convolution and pooling. Plus, ResNet has these 'shortcut' connections that skip some layers, making it easier for the network to learn and preventing it from getting stuck during training and making the network deeper, which improves the performance of the net.



```
# Load architecture
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np


print('Load model')

# Load the pre-trained ResNet50 model without the top classification layer
resnet_base = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze the weights of the pre-trained layers
for layer in resnet_base.layers:
    layer.trainable = False

# Create a new model by adding custom classification layers on top of the ResNet base

model = Sequential()
model.add(resnet_base)
model.add(GlobalAveragePooling2D())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(len(categories), activation='softmax'))

model.summary()
```

Figure 15: ResNet50 pre-trained from transfer learning

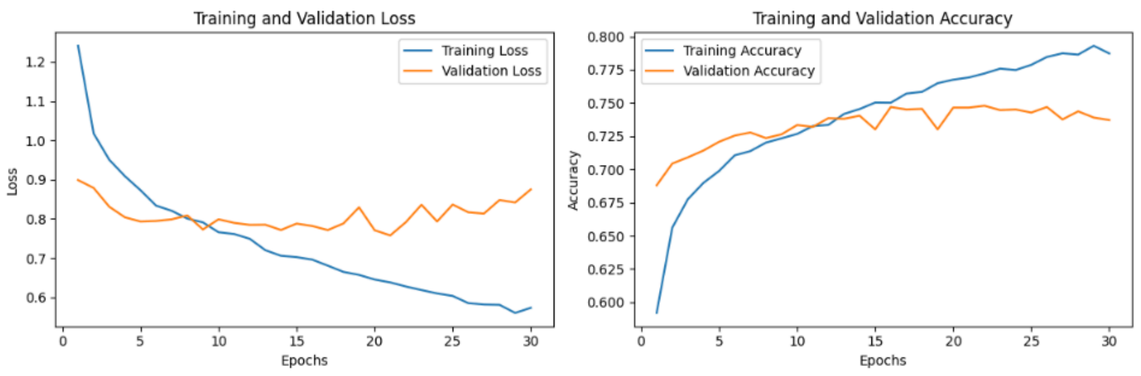


Figure 16: ResNet50 validation loss and accuracy metric

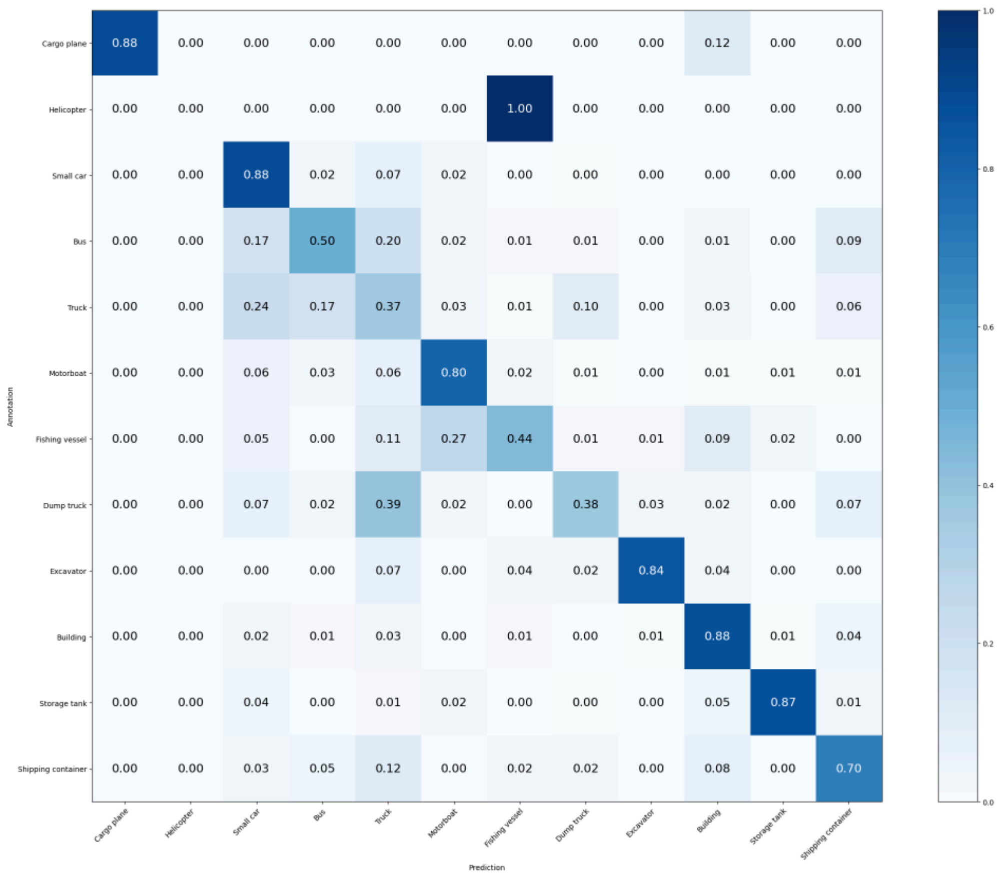


Figure 17: ResNet50 confusion matrix

```
Mean Accuracy: 72.789%
Mean Recall: 62.763%
Mean Precision: 64.567%
> Cargo plane: Recall: 87.952% Precision: 98.649% Specificity: 99.961% Dice: 92.994%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Small car: Recall: 88.296% Precision: 71.429% Specificity: 91.993% Dice: 78.972%
> Bus: Recall: 49.587% Precision: 58.252% Specificity: 96.406% Dice: 53.571%
> Truck: Recall: 37.049% Precision: 36.928% Specificity: 91.717% Dice: 36.989%
> Motorboat: Recall: 79.695% Precision: 84.865% Specificity: 97.501% Dice: 82.199%
> Fishing vessel: Recall: 44.086% Precision: 64.062% Specificity: 99.095% Dice: 52.229%
> Dump truck: Recall: 37.705% Precision: 53.488% Specificity: 98.408% Dice: 44.231%
> Excavator: Recall: 84.211% Precision: 82.759% Specificity: 99.612% Dice: 83.478%
> Building: Recall: 87.638% Precision: 89.792% Specificity: 97.420% Dice: 88.702%
> Storage tank: Recall: 87.243% Precision: 95.928% Specificity: 99.624% Dice: 91.379%
> Shipping container: Recall: 69.697% Precision: 38.655% Specificity: 97.158% Dice: 49.730%
```

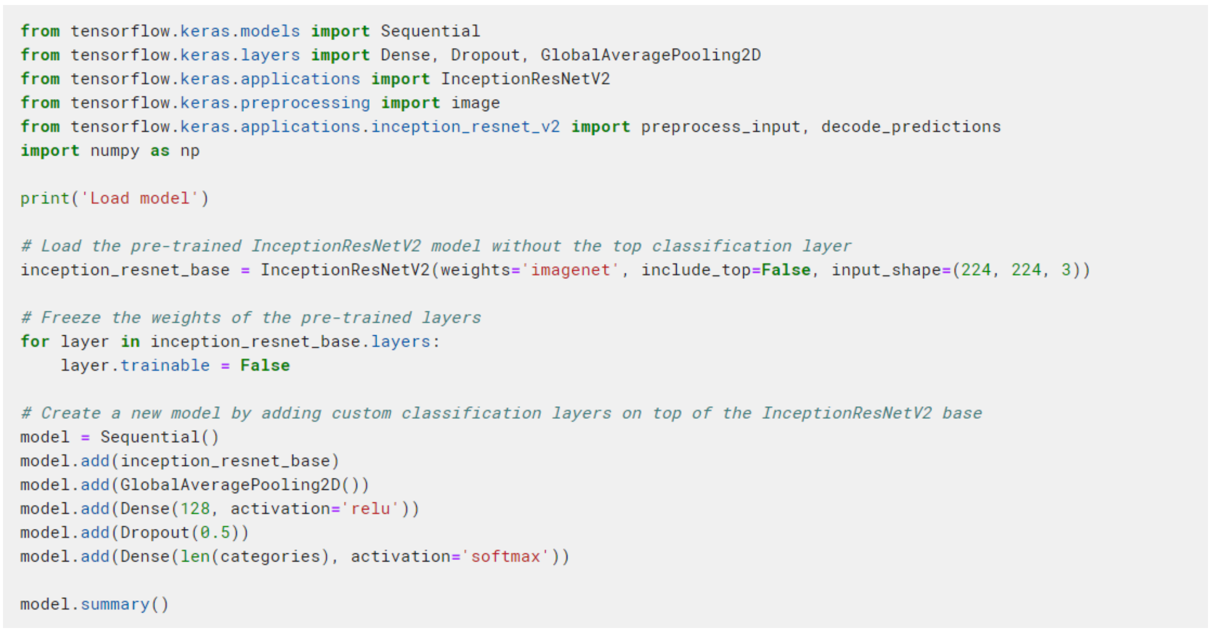
Figure 18: ResNet50 testing performances

The previous figures are the performances of this neural network. As we can see, the results obtained and the efficiency in the metrics are much better than all the other structures tried before. But we find a problem in this structure that we share with the rest of the nets, the class “Helicopter” is not being classified correctly once. The confusion matrix helps us to see this issue. Therefore, the next step for building the final network is to treat this issue.

For these, we are going now to train to make an oversampling to that class. We studied that the number of images of this class is much lower than for the other classes. That's why we decided to increase the number of images of the class “Helicopter” for the whole training (train+validation) set.

4.3.4 InceptionResnetV2 with transfer learning

Before tackling the “Helicopter” issue, we will try another pre-trained model. In this case the InceptionResNetV2. The following code loads InceptionResNetV2 as the base model and adds custom classification layers on top of it.



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
from tensorflow.keras.applications import InceptionResNetV2
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.inception_resnet_v2 import preprocess_input, decode_predictions
import numpy as np

print('Load model')

# Load the pre-trained InceptionResNetV2 model without the top classification layer
inception_resnet_base = InceptionResNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze the weights of the pre-trained layers
for layer in inception_resnet_base.layers:
    layer.trainable = False

# Create a new model by adding custom classification layers on top of the InceptionResNetV2 base
model = Sequential()
model.add(inception_resnet_base)
model.add(GlobalAveragePooling2D())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(len(categories), activation='softmax'))

model.summary()
```

Figure 19: InceptionResNetV2 code

After 20 epochs we start seeing how the model is not performing as expected. The loss and accuracy values obtained are very low compared with the other pre-trained models.

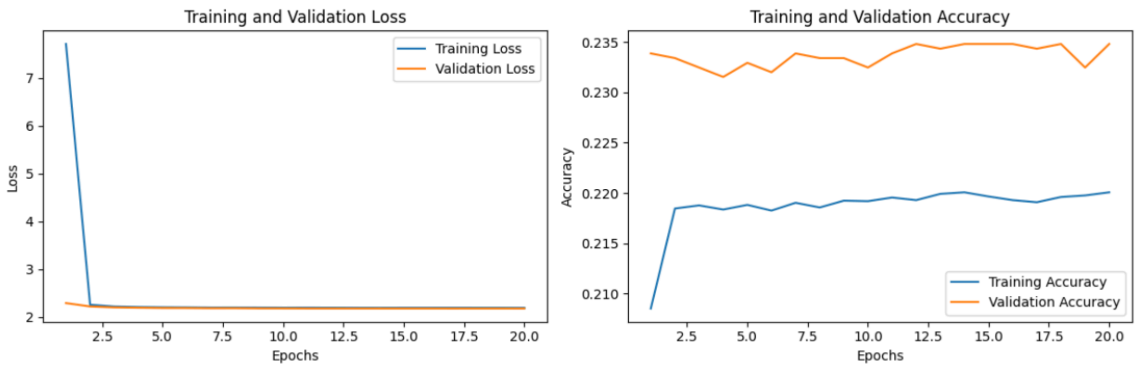


Figure 20: InceptionResNetV2 validation loss and accuracy metric

This statement is strengthened when we see the confusion matrix from the following figure. In this case, our model consistently predicts only one class (building) for nearly all predictions. There could be several reasons behind this behavior, limited model capacity, underfitting or even due to the unbalanced classes.

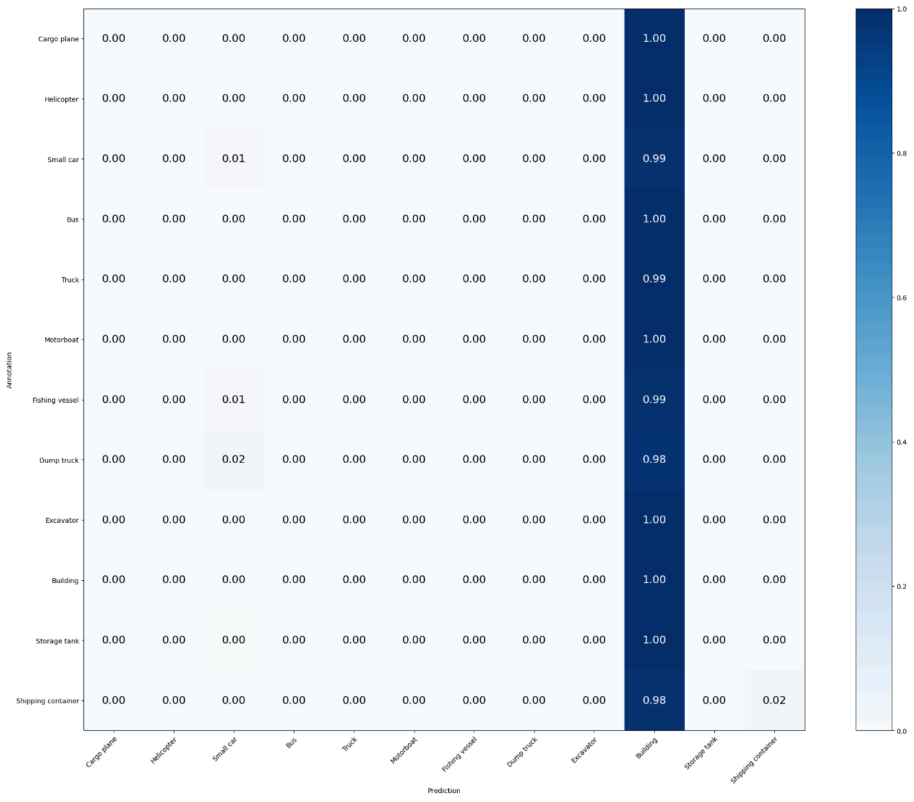
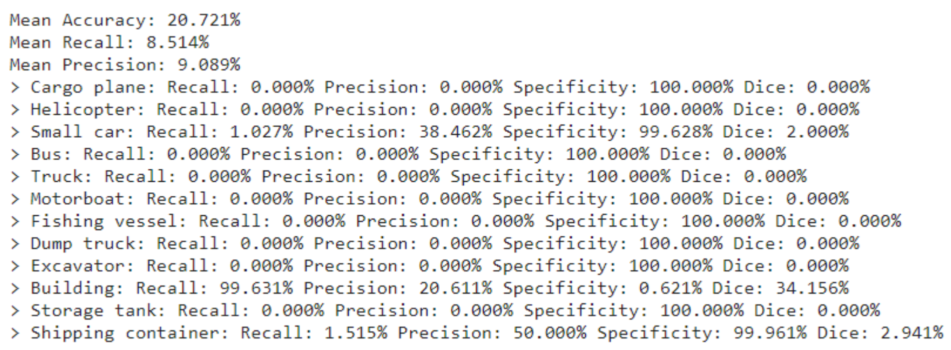


Figure 21: InceptionResNetV2 confusion matrix

Finally, we can see in the testing report how our model did not go as expected. Despite its reputation as a powerful architecture, the InceptionResNetV2 model struggled to generalize effectively to our dataset.



```
Mean Accuracy: 20.721%
Mean Recall: 8.514%
Mean Precision: 9.089%
> Cargo plane: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Small car: Recall: 1.027% Precision: 38.462% Specificity: 99.628% Dice: 2.000%
> Bus: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Truck: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Motorboat: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Fishing vessel: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Dump truck: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Excavator: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Building: Recall: 99.631% Precision: 20.611% Specificity: 0.621% Dice: 34.156%
> Storage tank: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Shipping container: Recall: 1.515% Precision: 50.000% Specificity: 99.961% Dice: 2.941%
```

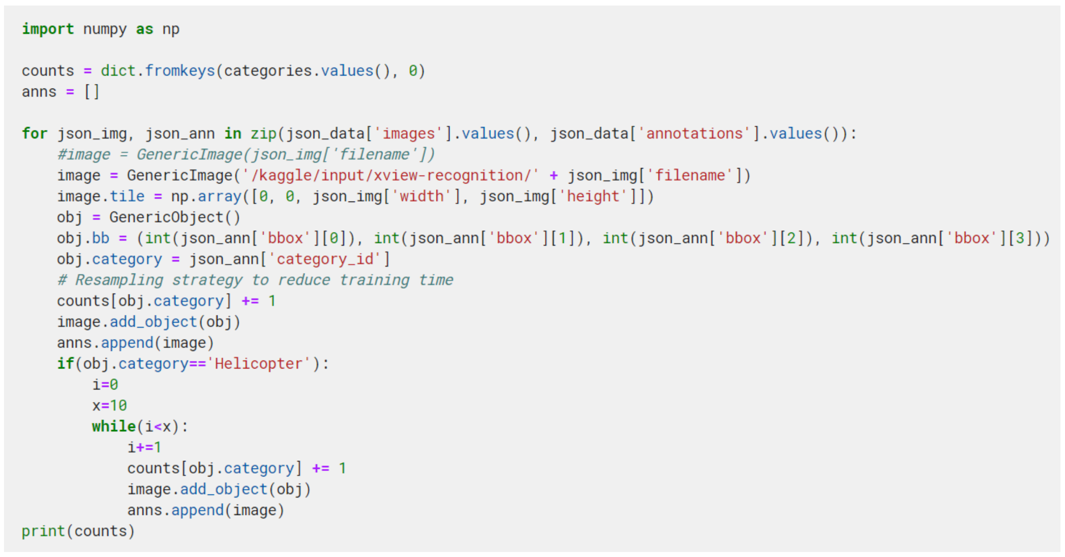
Figure 22: InceptionResNetV2 testing performances

In conclusion, we observed that the model tended to predict only one class indicating a significant bias in its predictions. This could be because deep neural networks, especially complex ones like InceptionResNetV2, require a large amount of data to generalize well. If the training dataset is relatively small or unbalanced, the complex model like InceptionResNetV2 may struggle to learn meaningful representations compared to ResNet50, which is less complex. Therefore, after careful consideration and evaluation, we have decided to continue with the ResNet50 architecture for our task.

4.4 Oversampling

4.4.1 Oversampling the “Helicopter” class

Oversampling is essential for addressing class imbalance in datasets. It ensures that all classes receive sufficient representation during model training. By artificially increasing the number of instances in minority classes, oversampling prevents models from becoming biased towards the majority class, leading to more balanced and accurate predictions. In this case we will be oversampling the helicopter and multiplying the number of images by 10 as seen in the following code.



```python
import numpy as np

counts = dict.fromkeys(categories.values(), 0)
anns = []

for json_img, json_ann in zip(json_data['images'].values(), json_data['annotations'].values()):
    #image = GenericImage(json_img['filename'])
    image = GenericImage('/kaggle/input/xview-recognition/' + json_img['filename'])
    image.tile = np.array([0, 0, json_img['width'], json_img['height']])
    obj = GenericObject()
    obj.bb = (int(json_ann['bbox'][0]), int(json_ann['bbox'][1]), int(json_ann['bbox'][2]), int(json_ann['bbox'][3]))
    obj.category = json_ann['category_id']
    # Resampling strategy to reduce training time
    counts[obj.category] += 1
    image.add_object(obj)
    anns.append(image)
    if(obj.category=='Helicopter'):
        i=0
        x=10
        while(i<x):
            i+=1
            counts[obj.category] += 1
            image.add_object(obj)
            anns.append(image)
print(counts)
```

Figure 23: Oversampling code for helicopter

Now we will train our ResNet50 model with this technique to see if we can improve the prediction accuracy in helicopters.

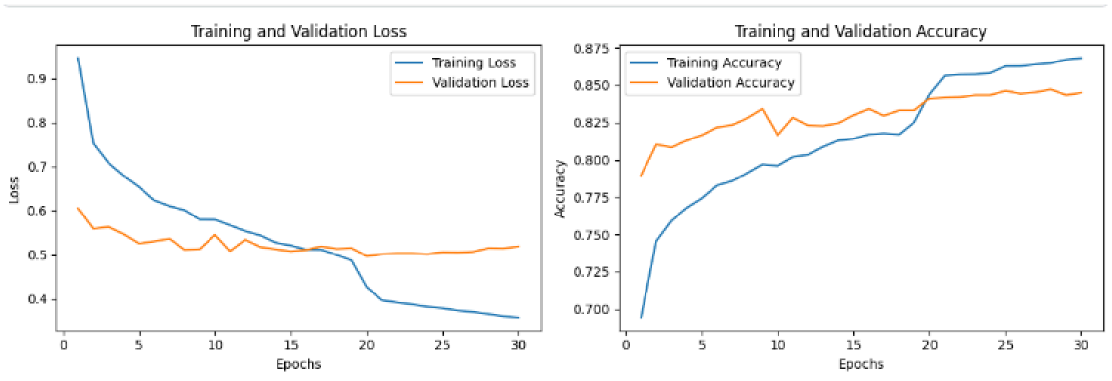


Figure 24: Oversampling validation loss and accuracy metric

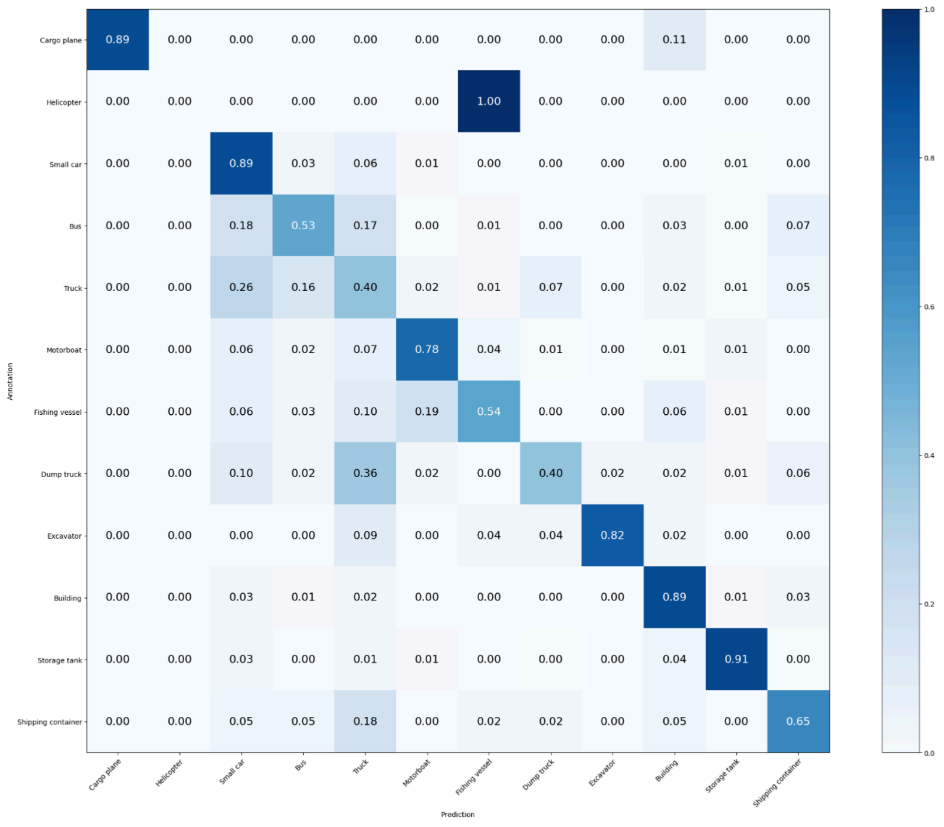
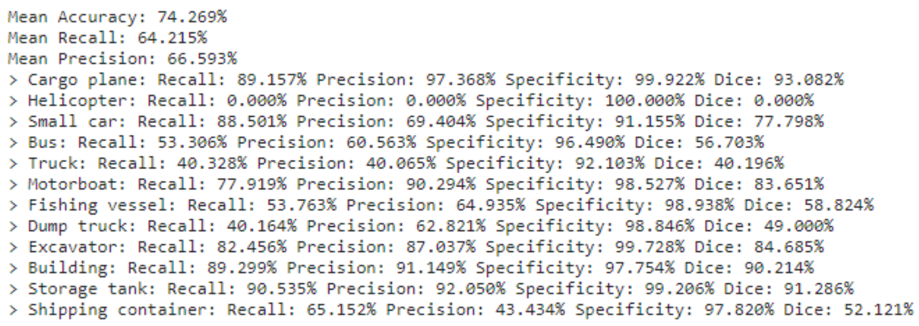


Figure 25: Oversampling confusion matrix



```
Mean Accuracy: 74.269%
Mean Recall: 64.215%
Mean Precision: 66.593%
> Cargo plane: Recall: 89.157% Precision: 97.368% Specificity: 99.922% Dice: 93.082%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 100.000% Dice: 0.000%
> Small car: Recall: 88.501% Precision: 69.404% Specificity: 91.155% Dice: 77.798%
> Bus: Recall: 53.306% Precision: 60.563% Specificity: 96.490% Dice: 56.703%
> Truck: Recall: 40.328% Precision: 40.065% Specificity: 92.103% Dice: 40.196%
> Motorboat: Recall: 77.919% Precision: 90.294% Specificity: 98.527% Dice: 83.651%
> Fishing vessel: Recall: 53.763% Precision: 64.935% Specificity: 98.938% Dice: 58.824%
> Dump truck: Recall: 40.164% Precision: 62.821% Specificity: 98.846% Dice: 49.000%
> Excavator: Recall: 82.456% Precision: 87.037% Specificity: 99.728% Dice: 84.685%
> Building: Recall: 89.299% Precision: 91.149% Specificity: 97.754% Dice: 90.214%
> Storage tank: Recall: 90.535% Precision: 92.050% Specificity: 99.206% Dice: 91.286%
> Shipping container: Recall: 65.152% Precision: 43.434% Specificity: 97.820% Dice: 52.121%
```

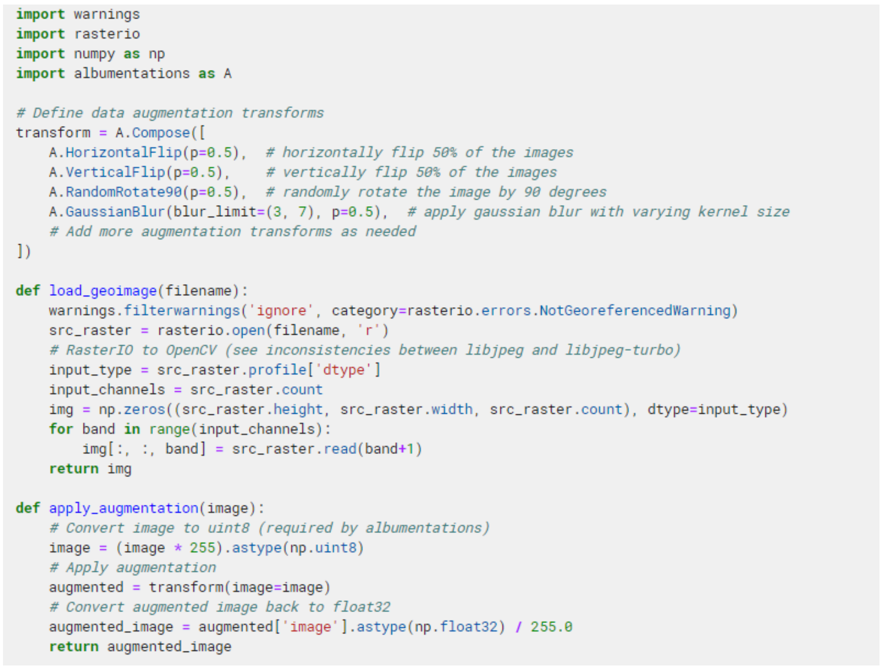
Figure 26: Oversampling testing performances

After implementing the oversampling technique to address the imbalance in the "helicopter" class, we unfortunately found that the issue persisted in our predictions. Despite our efforts to increase the representation of this class in the dataset, the model still struggled to make accurate predictions for “helicopter” instances. Recognizing the limitations of oversampling alone, we decided to explore alternative approaches, such as data augmentation, to further enhance the model's ability to learn from the available data. By augmenting the dataset with auto-generated samples that are similar to our original samples, we aim to provide the model with a more diverse and comprehensive set of examples, enabling it to better capture the issues of the "helicopter" class.

4.4.2 Data Augmentation

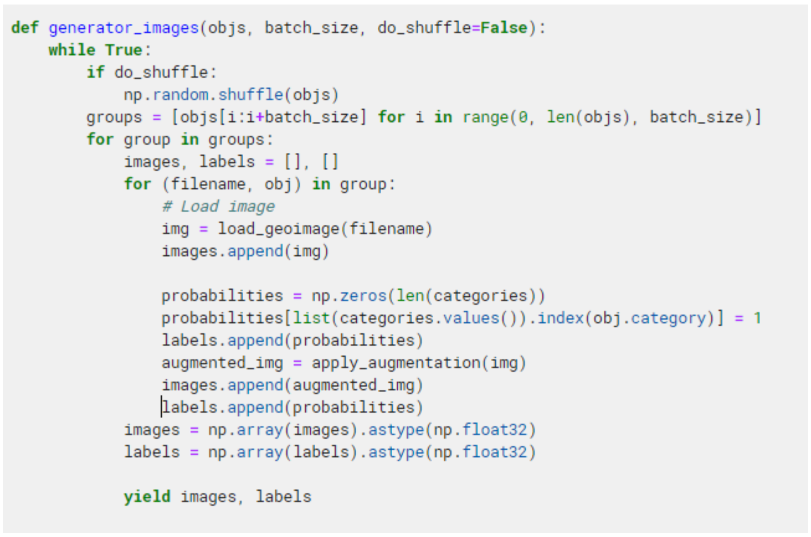
Data augmentation involves applying transformations like rotations, flips, and scaling to existing data, creating new samples with subtle variations. By diversifying the dataset, it helps prevent overfitting, improves model generalization, and is especially useful for addressing limited data or class imbalances. Overall, data augmentation enhances the robustness and performance of machine learning models across different tasks and domains

In this case, we are going to apply the data augmentation to all the datasets, not just the helicopter, with the following code. Our technique will consist of horizontally flipping the image, vertically flipping, rotating the image 90 degrees left of rate and finally applying the gaussian blur, all of these steps will have a probability of 50% to be applied to the image.



```python
import warnings
import rasterio
import numpy as np
import albumentations as A

# Define data augmentation transforms
transform = A.Compose([
    A.HorizontalFlip(p=0.5),  # horizontally flip 50% of the images
    A.VerticalFlip(p=0.5),    # vertically flip 50% of the images
    A.RandomRotate90(p=0.5),  # randomly rotate the image by 90 degrees
    A.GaussianBlur(blur_limit=(3, 7), p=0.5),  # apply gaussian blur with varying kernel size
    # Add more augmentation transforms as needed
])

def load_geoimage(filename):
    warnings.filterwarnings('ignore', category=rasterio.errors.NotGeoreferencedWarning)
    src_raster = rasterio.open(filename, 'r')
    # RasterIO to OpenCV (see inconsistencies between libjpeg and libjpeg-turbo)
    input_type = src_raster.profile['dtype']
    input_channels = src_raster.count
    img = np.zeros((src_raster.height, src_raster.width, src_raster.count), dtype=input_type)
    for band in range(input_channels):
        img[:, :, band] = src_raster.read(band+1)
    return img

def apply_augmentation(image):
    # Convert image to uint8 (required by albumentations)
    image = (image * 255).astype(np.uint8)
    # Apply augmentation
    augmented = transform(image=image)
    # Convert augmented image back to float32
    augmented_image = augmented['image'].astype(np.float32) / 255.0
    return augmented_image
```

Figure 27: Data augmentation code (I)



```python
def generator_images(objs, batch_size, do_shuffle=False):
    while True:
        if do_shuffle:
            np.random.shuffle(objs)
        groups = [objs[i:i+batch_size] for i in range(0, len(objs), batch_size)]
        for group in groups:
            images, labels = [], []
            for (filename, obj) in group:
                # Load image
                img = load_geoimage(filename)
                images.append(img)

                probabilities = np.zeros(len(categories))
                probabilities[list(categories.values()).index(obj.category)] = 1
                labels.append(probabilities)
                augmented_img = apply_augmentation(img)
                images.append(augmented_img)
                labels.append(probabilities)
            images = np.array(images).astype(np.float32)
            labels = np.array(labels).astype(np.float32)

            yield images, labels
```

Figure 28: Data augmentation code (II)

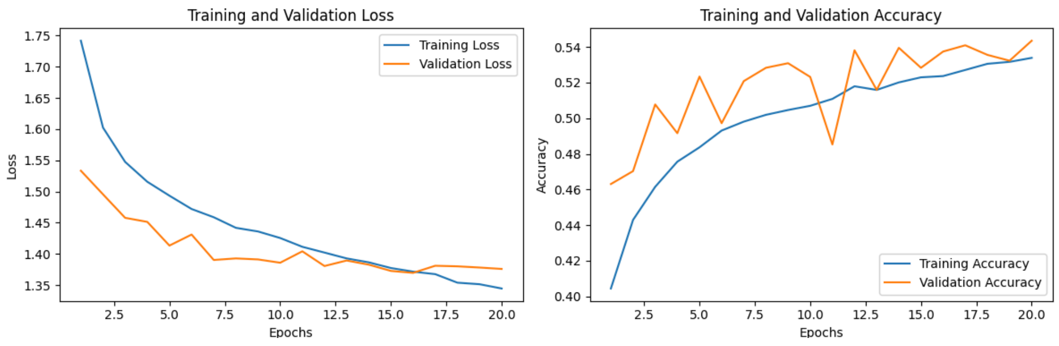


Figure 29: Data augmentation validation loss and accuracy metric

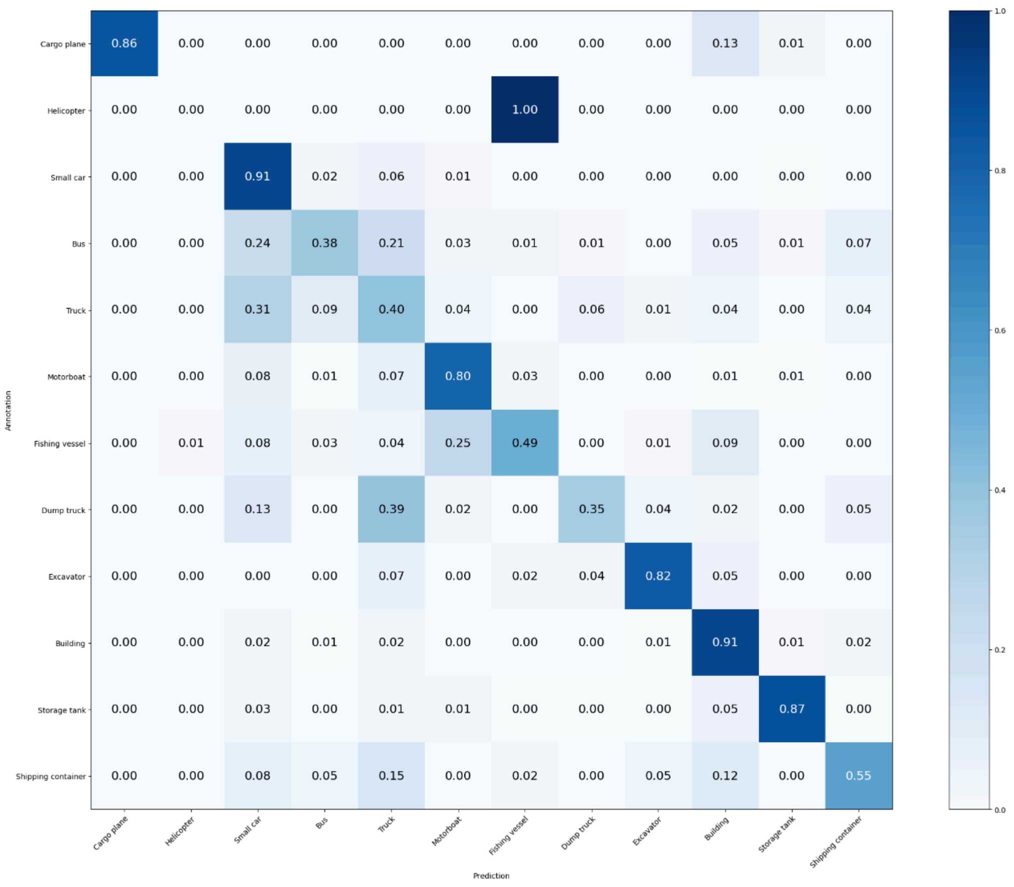
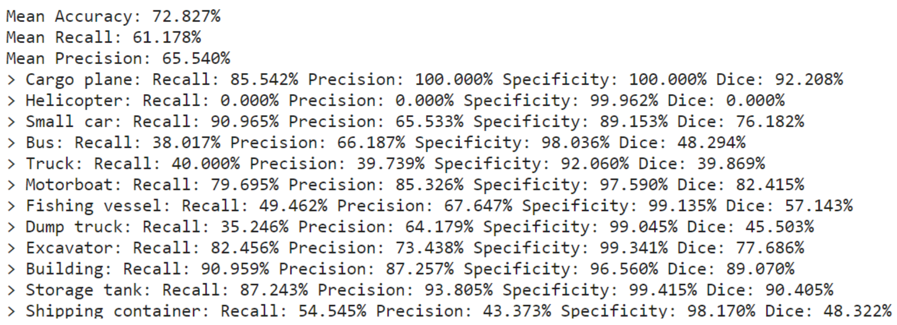


Figure 30: Data augmentation confusion matrix



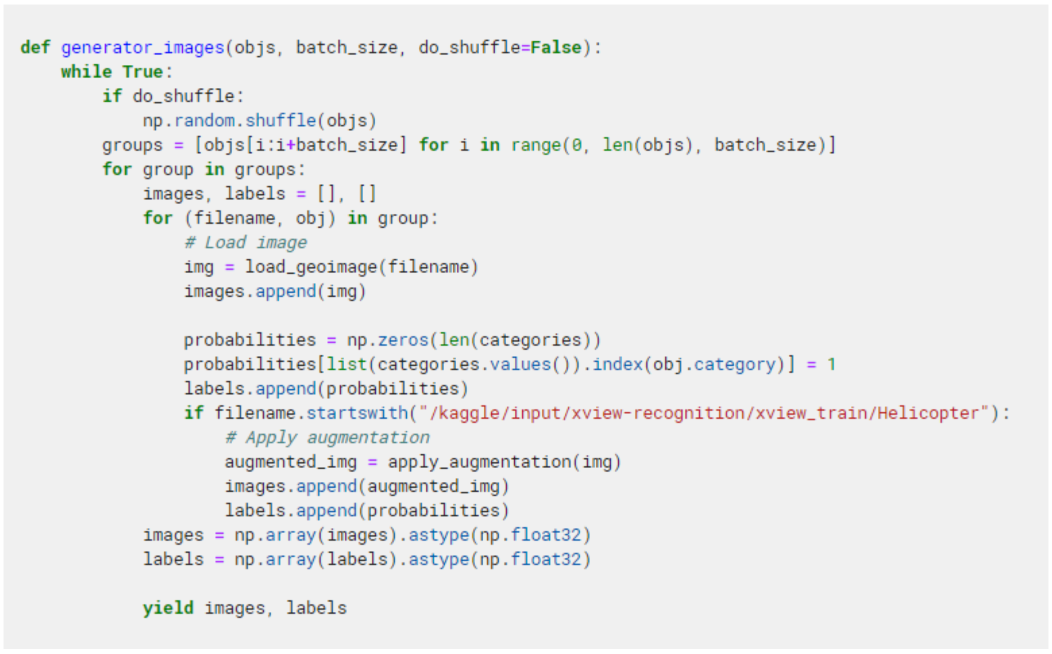
```
Mean Accuracy: 72.827%
Mean Recall: 61.178%
Mean Precision: 65.540%
> Cargo plane: Recall: 85.542% Precision: 100.000% Specificity: 100.000% Dice: 92.208%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 99.962% Dice: 0.000%
> Small car: Recall: 90.965% Precision: 65.533% Specificity: 89.153% Dice: 76.182%
> Bus: Recall: 38.017% Precision: 66.187% Specificity: 98.036% Dice: 48.294%
> Truck: Recall: 40.000% Precision: 39.739% Specificity: 92.060% Dice: 39.869%
> Motorboat: Recall: 79.695% Precision: 85.326% Specificity: 97.590% Dice: 82.415%
> Fishing vessel: Recall: 49.462% Precision: 67.647% Specificity: 99.135% Dice: 57.143%
> Dump truck: Recall: 35.246% Precision: 64.179% Specificity: 99.045% Dice: 45.503%
> Excavator: Recall: 82.456% Precision: 73.438% Specificity: 99.341% Dice: 77.686%
> Building: Recall: 90.959% Precision: 87.257% Specificity: 96.560% Dice: 89.070%
> Storage tank: Recall: 87.243% Precision: 93.805% Specificity: 99.415% Dice: 90.405%
> Shipping container: Recall: 54.545% Precision: 43.373% Specificity: 98.170% Dice: 48.322%
```

Figure 31: Data augmentation testing results

In our case, implementing data augmentation led to an increase in training time without a corresponding improvement in model accuracy. Despite the augmented dataset providing a broader range of examples, the model's performance did not significantly benefit from the additional training data. As a result, our next strategy involves merging oversampling and data augmentation, with a particular emphasis on the "helicopter" class. By directing these methods toward the underrepresented class, our goal is to furnish the model with a well-rounded and varied collection of samples pertinent to helicopters. This tactic capitalizes on the advantages of oversampling, boosting the presence of the minority class, and data augmentation, which injects diversity and guards against overfitting. By strategically intertwining these techniques, we anticipate refining the model's capacity to precisely categorize helicopter instances while mitigating any adverse effects on training duration.

4.4.3 Oversampling and Data Augmentation for Helicopter

In this stage we will combine both techniques previously used to see if we can solve the issue with the prediction accuracy of the helicopter class. To combine them we will be using the previous code but with a slight modification in the data augmentation to only apply it to the helicopter images.



```python
def generator_images(objs, batch_size, do_shuffle=False):
    while True:
        if do_shuffle:
            np.random.shuffle(objs)
        groups = [objs[i:i+batch_size] for i in range(0, len(objs), batch_size)]
        for group in groups:
            images, labels = [], []
            for (filename, obj) in group:
                # Load image
                img = load_geoimage(filename)
                images.append(img)

                probabilities = np.zeros(len(categories))
                probabilities[list(categories.values()).index(obj.category)] = 1
                labels.append(probabilities)
                if filename.startswith("/kaggle/input/xview-recognition/xview_train/Helicopter"):
                    # Apply augmentation
                    augmented_img = apply_augmentation(img)
                    images.append(augmented_img)
                    labels.append(probabilities)
            images = np.array(images).astype(np.float32)
            labels = np.array(labels).astype(np.float32)

            yield images, labels
```

Figure 32: Oversampling and Data augmentation code

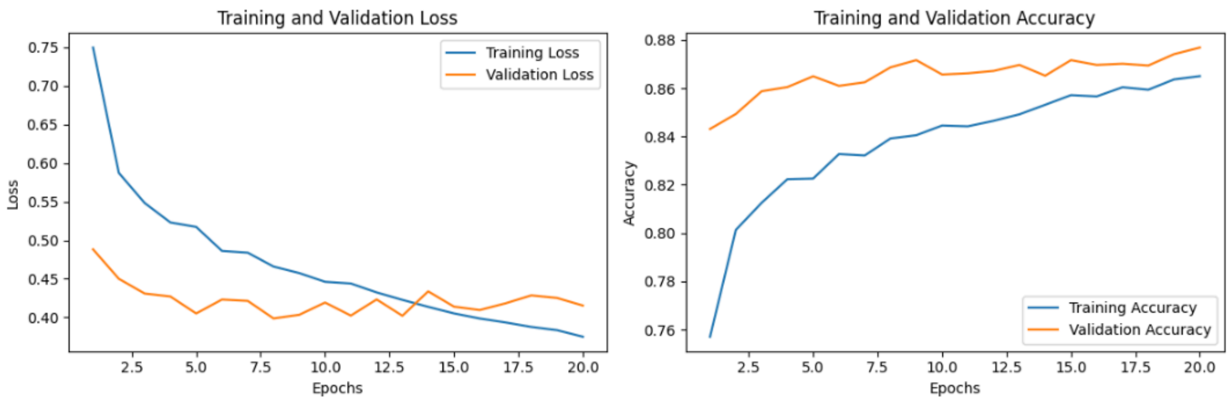


Figure 33: Oversampling and Data validation loss and accuracy metrics

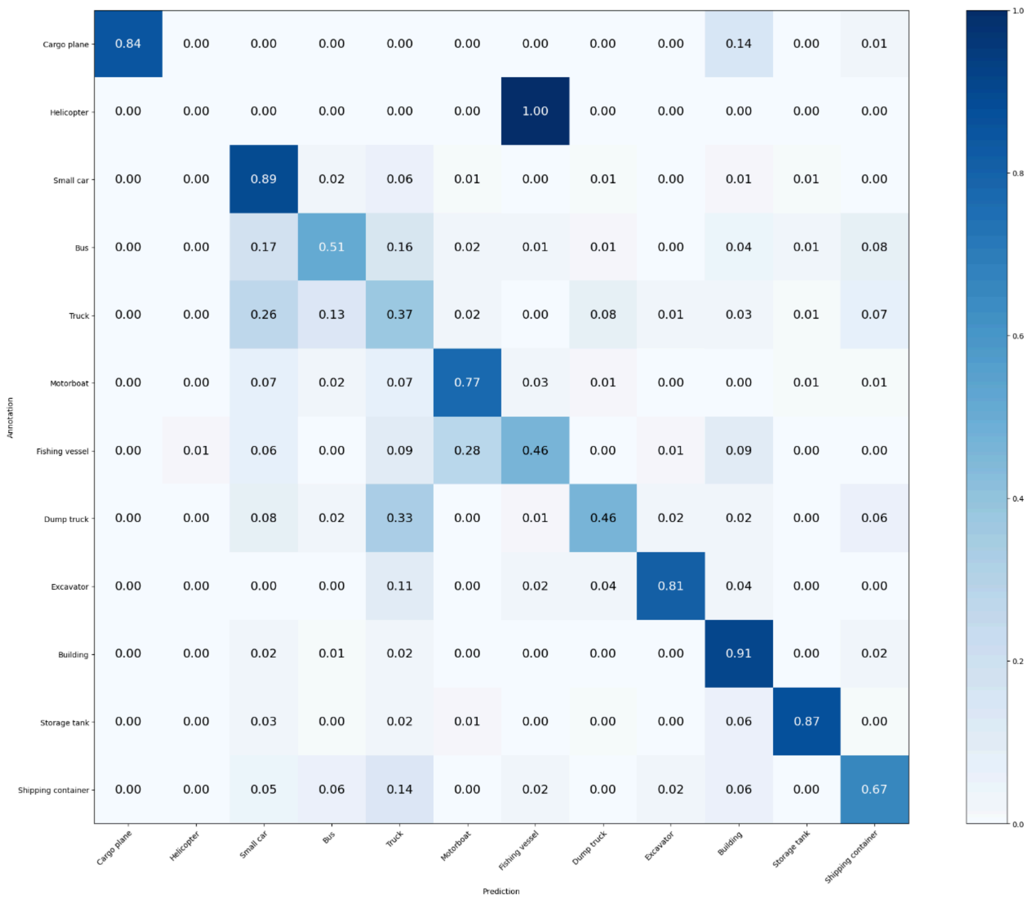
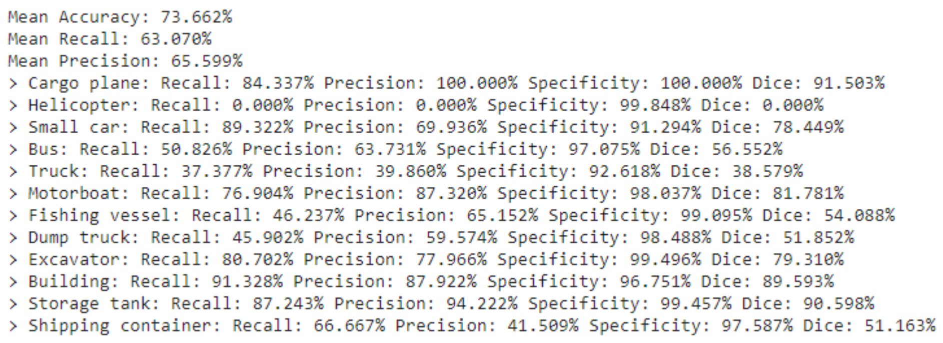


Figure 34: Oversampling and Data confusion matrix



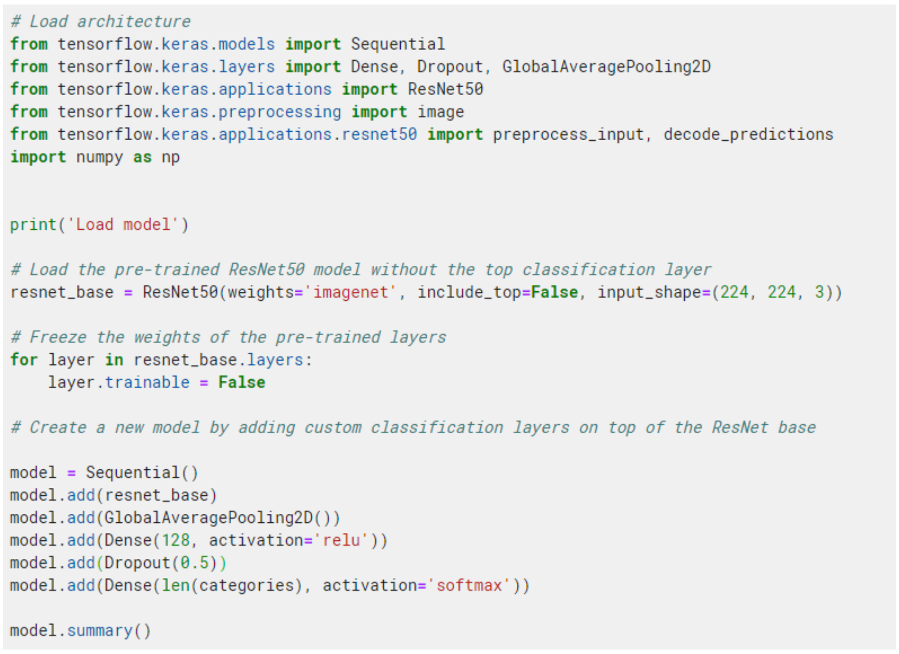
```
Mean Accuracy: 73.662%
Mean Recall: 63.070%
Mean Precision: 65.599%
> Cargo plane: Recall: 84.337% Precision: 100.000% Specificity: 100.000% Dice: 91.503%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 99.848% Dice: 0.000%
> Small car: Recall: 89.322% Precision: 69.936% Specificity: 91.294% Dice: 78.449%
> Bus: Recall: 50.826% Precision: 63.731% Specificity: 97.075% Dice: 56.552%
> Truck: Recall: 37.377% Precision: 39.860% Specificity: 92.618% Dice: 38.579%
> Motorboat: Recall: 76.904% Precision: 87.320% Specificity: 98.037% Dice: 81.781%
> Fishing vessel: Recall: 46.237% Precision: 65.152% Specificity: 99.095% Dice: 54.088%
> Dump truck: Recall: 45.902% Precision: 59.574% Specificity: 98.488% Dice: 51.852%
> Excavator: Recall: 80.702% Precision: 77.966% Specificity: 99.496% Dice: 79.310%
> Building: Recall: 91.328% Precision: 87.922% Specificity: 96.751% Dice: 89.593%
> Storage tank: Recall: 87.243% Precision: 94.222% Specificity: 99.457% Dice: 90.598%
> Shipping container: Recall: 66.667% Precision: 41.509% Specificity: 97.587% Dice: 51.163%
```

Figure 35: Oversampling and Data testing report

Regardless of our efforts in applying a combination of oversampling and data augmentation techniques to address the imbalance in the "helicopter" class, we regret to report that the issue persisted. The model's performance in the “helicopter” class did not meet our expectations, and the problem remains unresolved. As a result, we have decided to table this issue and leave it for future work. While we are unable to achieve a satisfactory solution at this time, we recognize the importance of revisiting this problem in the future and exploring alternative approaches that may yield better results. By leaving it as future work, we hope to encourage further investigation and experimentation to resolve the imbalance in the dataset and improve the performance of the model on this challenging task.

5. Final Results and Configurations Description

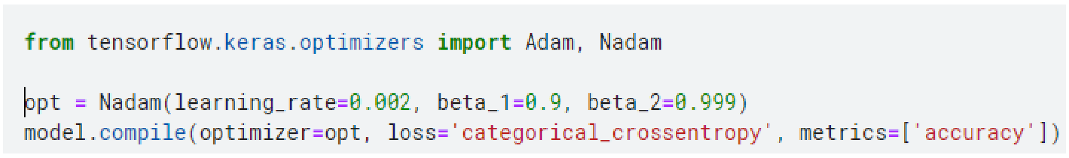
For our final configuration, we decided to go with the one that worked the best, which was the ResNet50. For this time we trained this net with 60 epochs just to have a consistent model and for the net to learn as much as possible. The net was pre-trained from transfer-learning and imported the following way:



```python
# Load architecture
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np


print('Load model')

# Load the pre-trained ResNet50 model without the top classification layer
resnet_base = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze the weights of the pre-trained layers
for layer in resnet_base.layers:
    layer.trainable = False

# Create a new model by adding custom classification layers on top of the ResNet base

model = Sequential()
model.add(resnet_base)
model.add(GlobalAveragePooling2D())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(len(categories), activation='softmax'))

model.summary()
```

Figure 36: Final configuration net

The optimizer used was the same as the previous structures, Nadam with these parameters, which was the best one found at the moment.



```python
from tensorflow.keras.optimizers import Adam, Nadam

opt = Nadam(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 37: Final optimizer

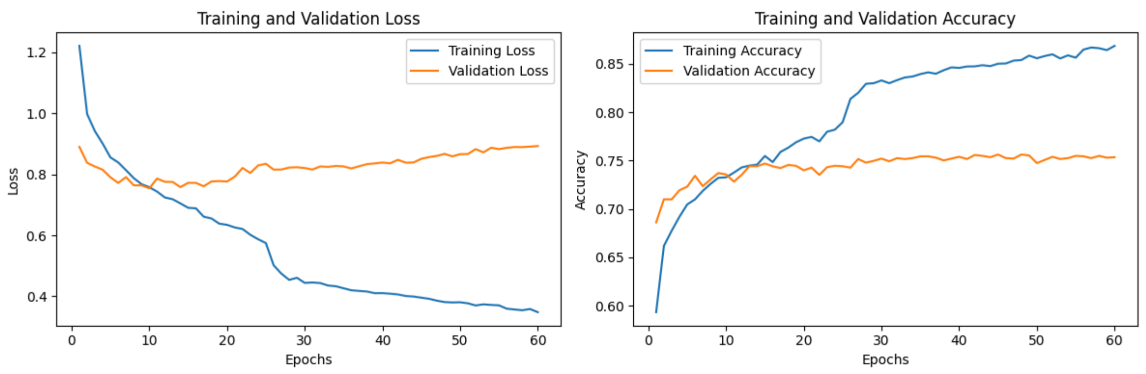


Figure 38: Loss function and validation accuracy metric

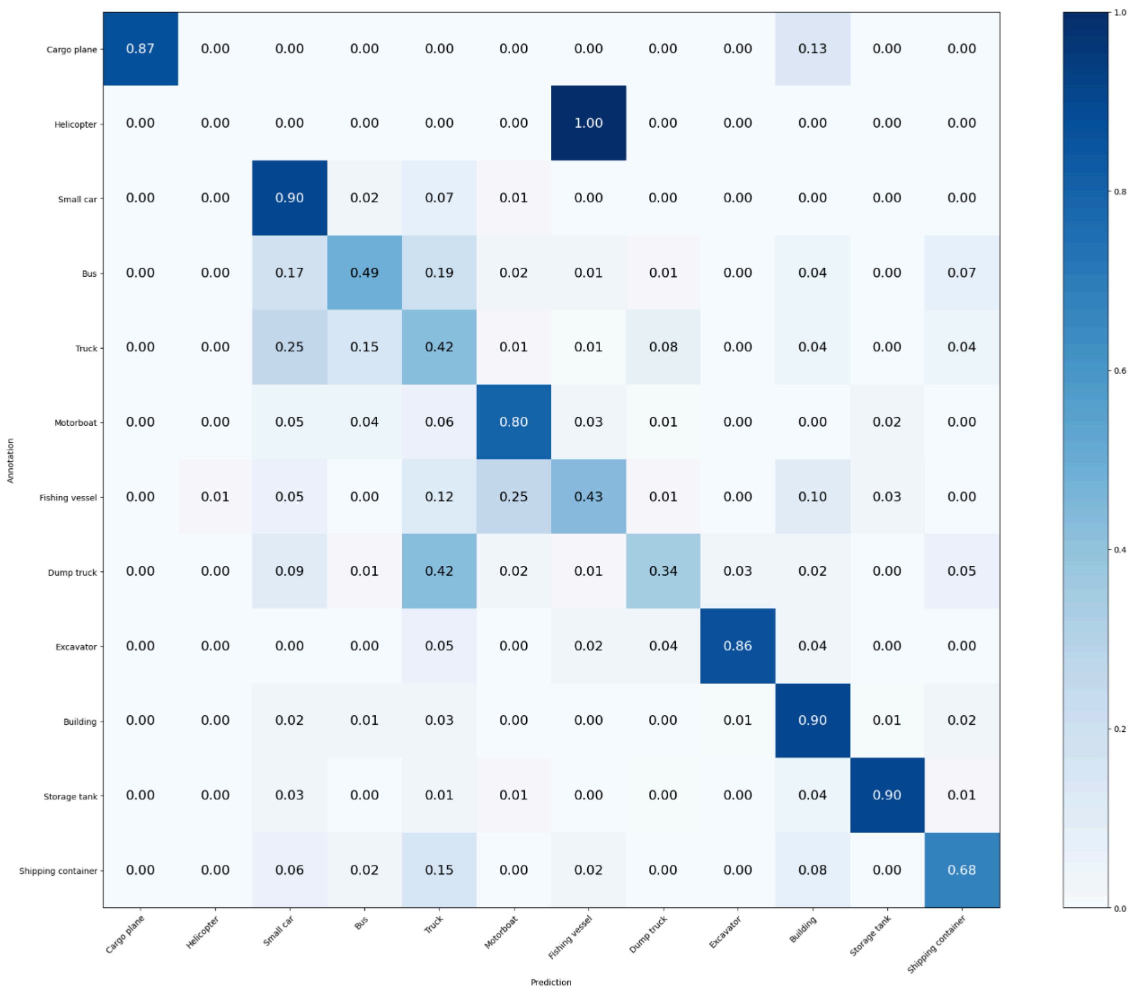


Figure 39: Final confusion matrix

```
Mean Accuracy: 74.080%
Mean Recall: 63.225%
Mean Precision: 65.898%
> Cargo plane: Recall: 86.747% Precision: 98.630% Specificity: 99.961% Dice: 92.308%
> Helicopter: Recall: 0.000% Precision: 0.000% Specificity: 99.962% Dice: 0.000%
> Small car: Recall: 89.528% Precision: 71.475% Specificity: 91.899% Dice: 79.490%
> Bus: Recall: 48.760% Precision: 59.596% Specificity: 96.657% Dice: 53.636%
> Truck: Recall: 41.967% Precision: 39.752% Specificity: 91.674% Dice: 40.829%
> Motorboat: Recall: 79.949% Precision: 88.235% Specificity: 98.126% Dice: 83.888%
> Fishing vessel: Recall: 43.011% Precision: 63.492% Specificity: 99.095% Dice: 51.282%
> Dump truck: Recall: 34.426% Precision: 53.846% Specificity: 98.567% Dice: 42.000%
> Excavator: Recall: 85.965% Precision: 84.483% Specificity: 99.651% Dice: 85.217%
> Building: Recall: 90.037% Precision: 88.889% Specificity: 97.086% Dice: 89.459%
> Storage tank: Recall: 90.123% Precision: 93.991% Specificity: 99.415% Dice: 92.017%
> Shipping container: Recall: 68.182% Precision: 48.387% Specificity: 98.132% Dice: 56.604%
```

Figure 40: Final results and accuracy

These are the performances and the matrix printed by the final structure. As we can see, the accuracy of the model improved, however, we decided to stop making oversampling and data augmentation of the class “Helicopter” because it didn't work correctly, that is why the performance for that class still remains the same.

6. Personal Experience

Our personal experience working on this project has been quite tough. Running computations on computers without GPUs has been slow and frustrating. Also, platforms like Google Colab and Kaggle often interrupt our work when the session expires, forcing us to start over. We wanted to try tweaking different settings to see if it improved our results, but we couldn't because of these issues. Furthermore, issues encountered with Cesvima prompted a change towards experimentation with Kaggle. Even though we tried using accelerators to speed things up, they didn't help much in the end.

However we found interesting how different types of neural networks work better or worse depending on many many parameters and characteristics of the net. We also were amazed by the capacity of these technologies, the accuracy of the net would never converge and could increase by a little amount each epoch if the computer would still be working on training and learning new patterns and features.

It was a tough problem to identify how and why many classes like “Helicopter” were misclassified and had to work on some investigation on how to treat with that to improve the accuracy of our models.

7. Conclusions

Using services like Kaggle has been a valuable experience for our project. While these platforms offer great resources for machine learning and data analysis, we have faced challenges with long computation times due to their limitations. However, despite these challenges, we found the project to be incredibly engaging and insightful.

Through our work, we have obtained a deeper understanding of image detection models and learned new techniques for tackling real-world problems. Despite the need for careful organization and patience when dealing with computation times, the project has provided us with valuable skills and knowledge that are directly applicable to the professional world.

We conclude after this study, that image classification needs a convolutional layer instead of dense layers. The performance of our structures, once we started using them, increased at a very high pace. They reduced the number of parameters and identified much better some patterns of the data. In comparison with this, we found interesting how popular architectures are working pretty well with the xView dataset. However, not all of them, but the ResNet50 showed a really good performance for this dataset.

To treat the unbalanced class “Helicopter” we tried to apply several ways to make this class more relevant and try to increase the number of instances classified correctly of this class. Although we tried many ways, there are so many more like smote, or undersampling, that could be used in future works related to neural nets.

Even though we tried many times and with different ways of making the “Helicopter” class being classified correctly, we didn't fulfill it, that is why we decided to finish the data augmentation technique just because it only increased the computational times when training the model.

In conclusion, while there were troubles to overcome, the experience of working in this project has been positive overall. We believe that future projects with different techniques could overcome some troubles we had in this. Finally, we are excited to apply the skills and insights gained from this project to future efforts.

8. References

[1] Keras: la API de alto nivel para TensorFlow. (n.d.-b). TensorFlow. https://www.tensorflow.org/guide/keras?hl=es

[2] XView. (n.d.). http://xviewdataset.org/

[3] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2014, September 17). Going Deeper with Convolutions. arXiv.org. https://arxiv.org/abs/1409.4842v1

[4] Boesch, G. (2024, March 12). VGG Very Deep Convolutional Networks (VGGNet) – What you need to know. viso.ai. https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/

[5] ResNet: The Basics and 3 ResNet Extensions. (2023, May 23). Datagen. https://datagen.tech/guides/computer-vision/resnet/#:~:text=Residual%20Network%20(ResNet)%20is%20a,or%20thousands%20of%20convolutional%20layers.

[6] ImageNet. (n.d.). https://www.image-net.org/