Open in app ↗

Sign up Sign in

Medium

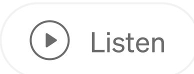
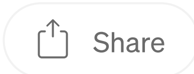
Search

AI for Classifying Pictures from the International Space Station

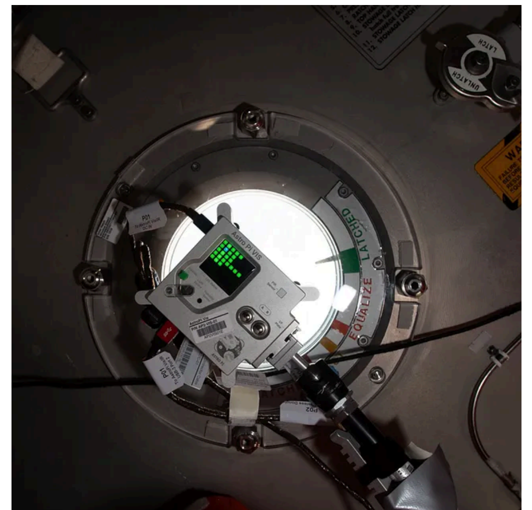
Is AI Capable of Recognizing Pictures Taken from the ISS? Which one is Better: CNN, DNN, or Pre-Trained Models?

Eva · Follow

17 min read · Apr 20, 2024

Have you ever wondered what it's like to see Earth from space? A few years ago, I had the incredible opportunity to participate in a space program that allowed me to control a Raspberry Pi on the International Space Station (ISS) during two and a half orbits. This tiny computer was equipped with a camera, capturing breathtaking images of our planet from its 400-kilometer vantage point, and an additional AI device could be used.



Raspberry Pi device ready to take pictures from space through the porthole.

I wasn't the only one collecting these spacefaring photos. To expand my dataset, I also incorporated publicly available images from other teams on Flickr. Now, with a collection of around 700 images, I decided to leverage the power of artificial intelligence (AI) to analyze them.

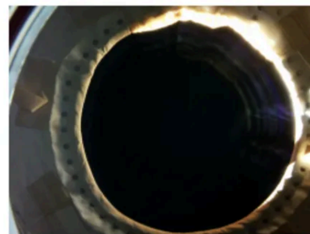
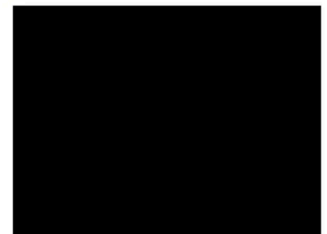
Here's the challenge: Since the ISS has limited storage, I wanted to focus on classifying the images into day and night categories. Nighttime photos, with little visibility, wouldn't be very interesting. Additionally, land masses hold more visual intrigue compared to vast oceans, offering glimpses of human structures and geographical features. Therefore, my goal was to develop an AI model that could not only distinguish day and night but also prioritize land over water for further analysis.

Preliminary importations:

```
!pip install --quiet gdown==4.5.4 --no-cache-dir
import os
import random

from PIL import Image as PILImage
from IPython.display import Image

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
from math import floor
import tensorflow as tf
from tensorflow import keras
from keras import layers
from sklearn.model_selection import train_test_split
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Input, Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D, Lambda
from keras import backend as K
from keras.optimizers import *
from keras.applications.vgg16 import VGG16
from keras.metrics import *
from keras.applications.vgg16 import preprocess_input
```

```
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras.layers.experimental.preprocessing import RandomFlip, Ranc
```

```
# Plotting examples of training data
# Plotting examples from our image data
_, ax = plt.subplots(1, 4, figsize=(15,60))  # to show 4 images side by side, m
ax[0].imshow(plt.imread(os.path.join(train_day_dir, "photo_407.jpg")))  # show
ax[0].axis('off')
ax[1].imshow(plt.imread(os.path.join(train_day_dir, "photo_164.jpg")))  # show
ax[1].axis('off')
ax[2].imshow(plt.imread(os.path.join(train_night_dir, "photo_341.jpg")))  # shc
ax[2].axis('off')
ax[3].imshow(plt.imread(os.path.join(train_night_dir, "photo_095.jpg")))  # shc
ax[3].axis('off')

plt.show()
```

Some of the Training images for day/night classification. Day with clouds, piece of land, ISS just after sunset, night among the 45-minute-long night

This is where deep learning comes in! To tackle the image classification task, I opted for Deep Neural Networks (DNNs) as my machine learning algorithm of choice. DNNs are a powerful type of artificial neural network inspired by the structure and function of the human brain. They consist of multiple interconnected layers: an input layer that receives the image data, hidden layers that process and extract features from the data, and an output layer that delivers the classification results (day/night or land/ocean in this case).

Within these layers, artificial neurons, often called “perceptrons,” act as the building blocks. Each neuron receives weighted inputs from the previous layer, performs a mathematical calculation, and generates an output. The connections between neurons and their associated weights are crucial for learning, as the network iteratively adjusts these weights to improve its classification accuracy.

DNNs excel at image recognition tasks due to their ability to learn complex patterns within image data. This makes them well-suited for analyzing the rich visual information captured in the space photos.

While I primarily focused on DNNs, I experimented with different architectures throughout my project. We'll delve deeper into these variations later, but for now, it's important to understand the core concept of DNNs and their potential for image classification.

Throughout this article, I'll explore how I trained various machine learning models to tackle this image classification challenge. We'll delve into their inner workings, analyze their effectiveness, and ultimately explore the exciting possibilities for future space photography endeavors.

Part 1: Day or night ?

A Simple Test for AI, a Big Step for me

Since classifying day and night images is a relatively straightforward task, I decided to use it as an initial test for myself and the AI. This step allows me to apply my newly acquired knowledge and gauge the model's capabilities. Here, I hope to see promising results before tackling more complex challenges.

1.1: Deep Neural Networks (DNN)

For image classification, I'll be employing a deep neural network (DNN) model built using the Python library *Keras*.

1.1.1 Dataset creation and preparation

To make them usable by the model, I'll assign labels to each picture and convert the data into a format the algorithm can understand. This involves splitting the dataset into separate folders for training and validation data, typically with a ratio of 80% for training and 20% for validation. The folder structure is shown below:

```
.
└── Train_x_AstroPi/
    ├── train/
    │   ├── day/
    │   │   └── *215 pictures*
    │   └── night/
    │       └── *122 pictures*
    └── validation/
        ├── day/
```

```
│   └── *47 pictures*
└── night/
    └── *30 pictures*
```

(Ascii made with nathan's tool)

1.1.2 Importing and splitting data

With labeled images organized, the next step is to import the data into Colab using libraries like TensorFlow or scikit-learn. We'll then split the data further within the code for training and validation.

```
!gdown 1PzvnbKLFf69bD67I6XylGsVWRgAK677K
!unzip -qq TRAIN_x_AstroPi.zip

# Select the directory where the data is saved
data_dir = 'TRAIN_x_AstroPi/'
# Set the path to the directory where the data is stored
train_directory = os.path.join(data_dir, 'train')
validation_directory = os.path.join(data_dir, 'validation')
# Setting the data directories for each class in the train set
train_day_dir = os.path.join(data_dir, 'train/day')
train_night_dir = os.path.join(data_dir, 'train/night')
# Setting the data directories for each class in the train set
validation_day_dir = os.path.join(data_dir, 'validation/day')
validation_night_dir = os.path.join(data_dir, 'validation/night')
```

Console returning:

```
Downloading...
From: https://drive.google.com/uc?id=1PzvnbKLFf69bD67I6XylGsVWRgAK677K
To: /content/TRAIN_x_AstroPi.zip
100% 1.94G/1.94G [00:35<00:00, 54.5MB/s]
```

Next, we will use `ImageDataGenerator()` to preprocess the data efficiently and quickly. The ImageDataGenerator class will automatically generate batches of images and labels to feed into the model during training, which makes the process memory-efficient.

```
# Load the training data using the ImageDataGenerator class
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    train_directory,
    target_size=(150, 110),
    batch_size=32,
    class_mode='binary'
)
```

```
Found 337 images belonging to 2 classes.
```

```
# Load the validation data using the ImageDataGenerator class
validation_datagen = ImageDataGenerator(rescale=1./255)
validation_generator = validation_datagen.flow_from_directory(
    validation_directory,
    target_size=(150, 110),
    batch_size=32,
    shuffle = False,
    class_mode='binary'
)
```

```
Found 77 images belonging to 2 classes.
```

1.1.3 Building the Neural Network

Then, I define the input shape of the data and build my Neural Network:

```
# Define the input shape for the model
input_shape = (150, 110, 3)
# This specifies that the input is an image with a height of 150 pixels,
# a width of 110 pixels, and 3 color channels (RGB).

# Initialize the model
model = Sequential()
# Flatten image and add input layer
model.add(Flatten(input_shape = input_shape))
# Add fully connected layer with 512 neurons and ReLU activation
model.add(Dense(512, activation='relu'))
# Add fully connected layer with 512 neurons and ReLU activation
model.add(Dense(512, activation='relu'))
```

```
    # Output layer
    model.add(Dense(1, activation='sigmoid'))
```

The `Sequential()` function in Keras is used to create a Neural Network which allows us to add layers to our neural network in a sequential manner.

The `Flatten()` function in Keras is used to convert data into a one-dimensional format. For an image, it adds all the pixels next to each other, in 1D.

Following the flattening layer, we add two fully connected layers, each with 512 neurons and ReLU activation functions. Then I determine the dimensions of the data. Fully connected layers are also known as dense layers, where each neuron in a layer is connected to every neuron in the previous layer.

Finally, we add an output layer with a single neuron and a sigmoid activation function. This output layer is responsible for producing the final output of our model, which in this case is a binary classification result (0 or 1) indicating the predicted class of the input image ('day' or 'night'). The sigmoid activation function is commonly used for binary classification tasks because it squashes the output values to the range [0, 1], representing the probability of belonging to one of the two classes.

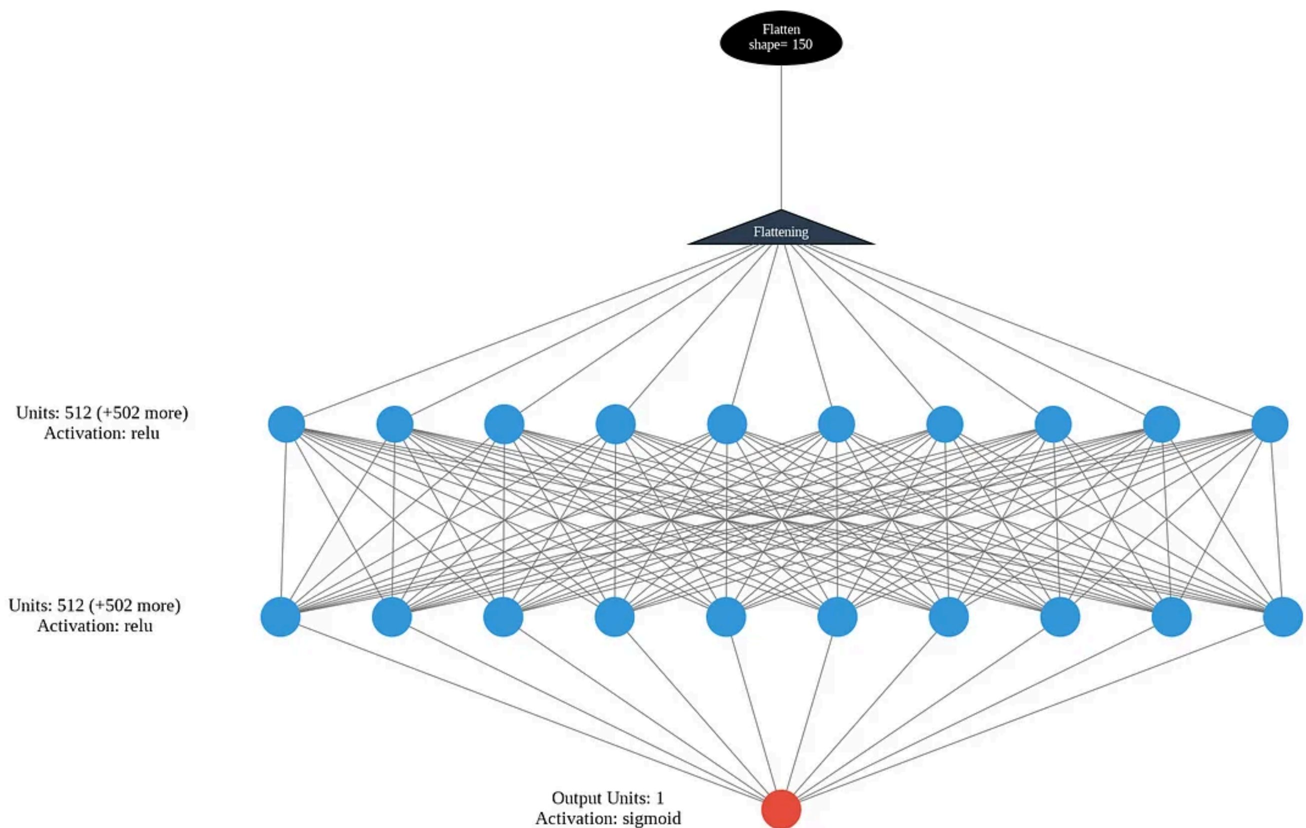
Then we use `model.summary()` to take a look at how many model parameters we have.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 49500)             0

 dense (Dense)               (None, 512)               25344512

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 1)                 513

=================================================================
Total params: 25607681 (97.69 MB)
Trainable params: 25607681 (97.69 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Screenshot of the Colab Notebook displaying, in particular, the number of trainable parameters: more than 25 million!

To have a different view of our DNN, we compute:

```
!pip install --quiet keras_visualizer
from keras_visualizer import visualizer
from IPython.display import Image

visualizer(model, file_name="model", file_format='png')
Image("model.png")
```



Each neuron in a Dense layer is connected to every other neuron in the next layer.

1.1.4 Training the Neural Network

We use `binary crossentropy` for the loss function, `optimizer='adam'` , and `metrics = ['accuracy']` as the evaluation metric.

Note: We use binary crossentropy for the loss since this is a classification problem (day or night classes).

```
# Compile the model
#opt = Adam(learning_rate = 0.1)
model.compile(optimizer='adam', loss='binary_crossentropy',
```

```
    metrics = ['accuracy'])

# Train the model using the generator
history = model.fit(
    train_generator,
    steps_per_epoch=10,
    epochs=8,
    validation_data=validation_generator,
    validation_steps=20
)
```

```
Epoch 1/8
10/10 [==============================] - ETA: 0s - loss: 0.4022 - accuracy: 0.9187WARNING:tensorflow:Your input ran out of data; inter
10/10 [==============================] - 72s 6s/step - loss: 0.4022 - accuracy: 0.9187 - val_loss: 0.4727 - val_accuracy: 0.9870
Epoch 2/8
10/10 [==============================] - 45s 4s/step - loss: 0.3246 - accuracy: 0.9967
Epoch 3/8
10/10 [==============================] - 42s 4s/step - loss: 0.3569 - accuracy: 0.9902
Epoch 4/8
10/10 [==============================] - 43s 4s/step - loss: 0.1695 - accuracy: 0.9934
Epoch 5/8
10/10 [==============================] - 42s 4s/step - loss: 0.3084 - accuracy: 0.9934
Epoch 6/8
10/10 [==============================] - 42s 4s/step - loss: 0.2918 - accuracy: 0.9934
Epoch 7/8
10/10 [==============================] - 44s 4s/step - loss: 0.2329 - accuracy: 0.9967
Epoch 8/8
10/10 [==============================] - 43s 4s/step - loss: 0.1624 - accuracy: 0.9967
```

High validation accuracy (99%)! There's a small error, but the model performs well overall. I'll fix it for Part 2.

1.1.4 Evaluate the Model

First, we make the predictions. Then we visualize them.

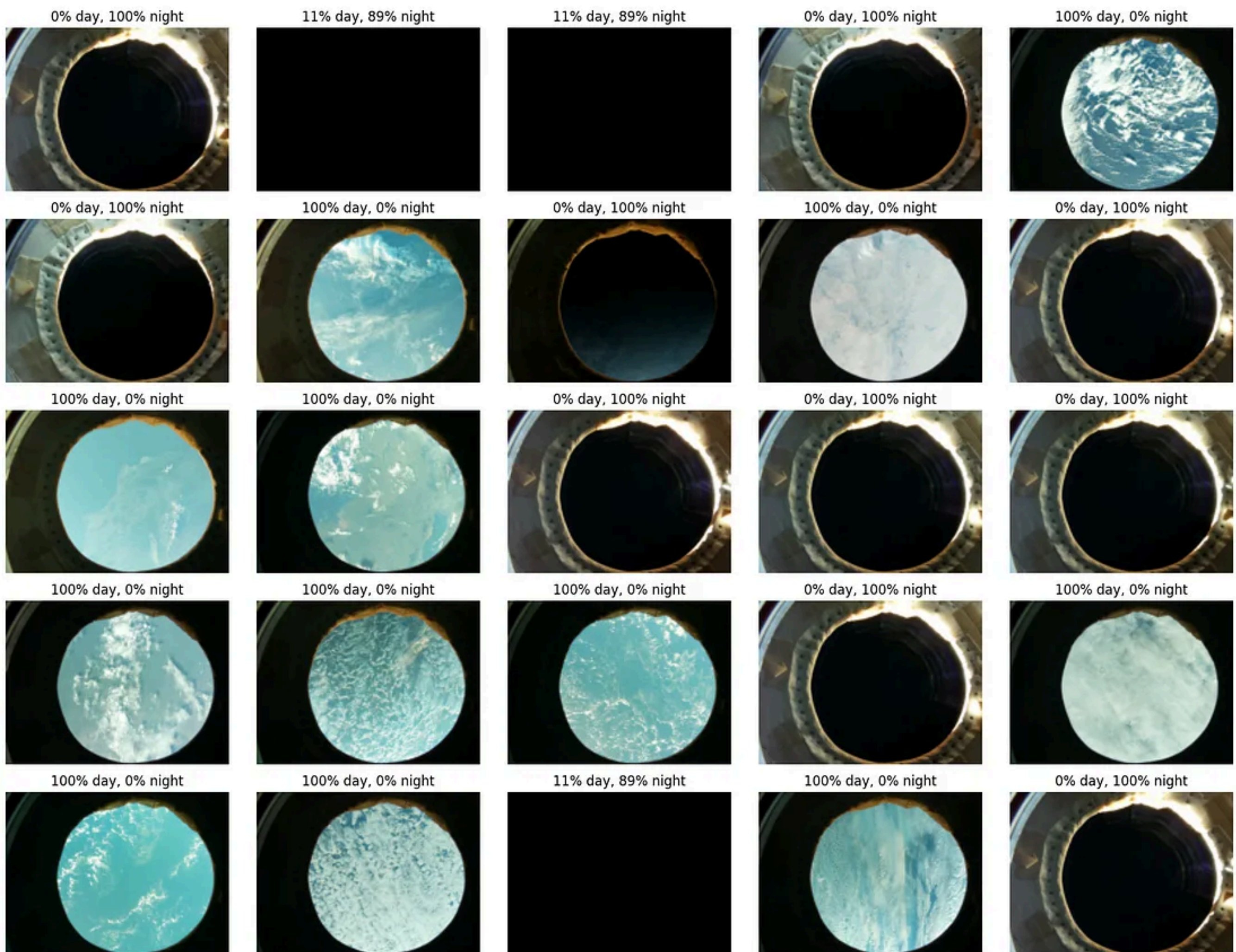
```
# Reset indices in generator
validation_generator.reset()

# Predict the class probabilities for the validation data
pred_probs = model.predict(validation_generator)
```

```
3/3 [==============================] - 11s 3s/step
```

Visualize the result by examining a sample of the predictions:

```
# Create grid for images
fig = plt.figure(figsize=(20, 20))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(5, 5),  # creates 2x2 grid of axes
                 axes_pad=0.35,  # pad between axes in inch.
                 )
```

```
validation_generator.reset()
for ax, i in zip(grid, random.sample(range(len(validation_generator.filenames))
    img = PILImage.open(os.path.join(validation_directory,validation_generator.
    img = img.resize((150,110))
    ax.imshow(img)
    ax.axis("off")
    ax.set_title("{:.0f}% day, {:.0f}% night".format(100*(1-pred_probs[i][0]),
plt.show()
```

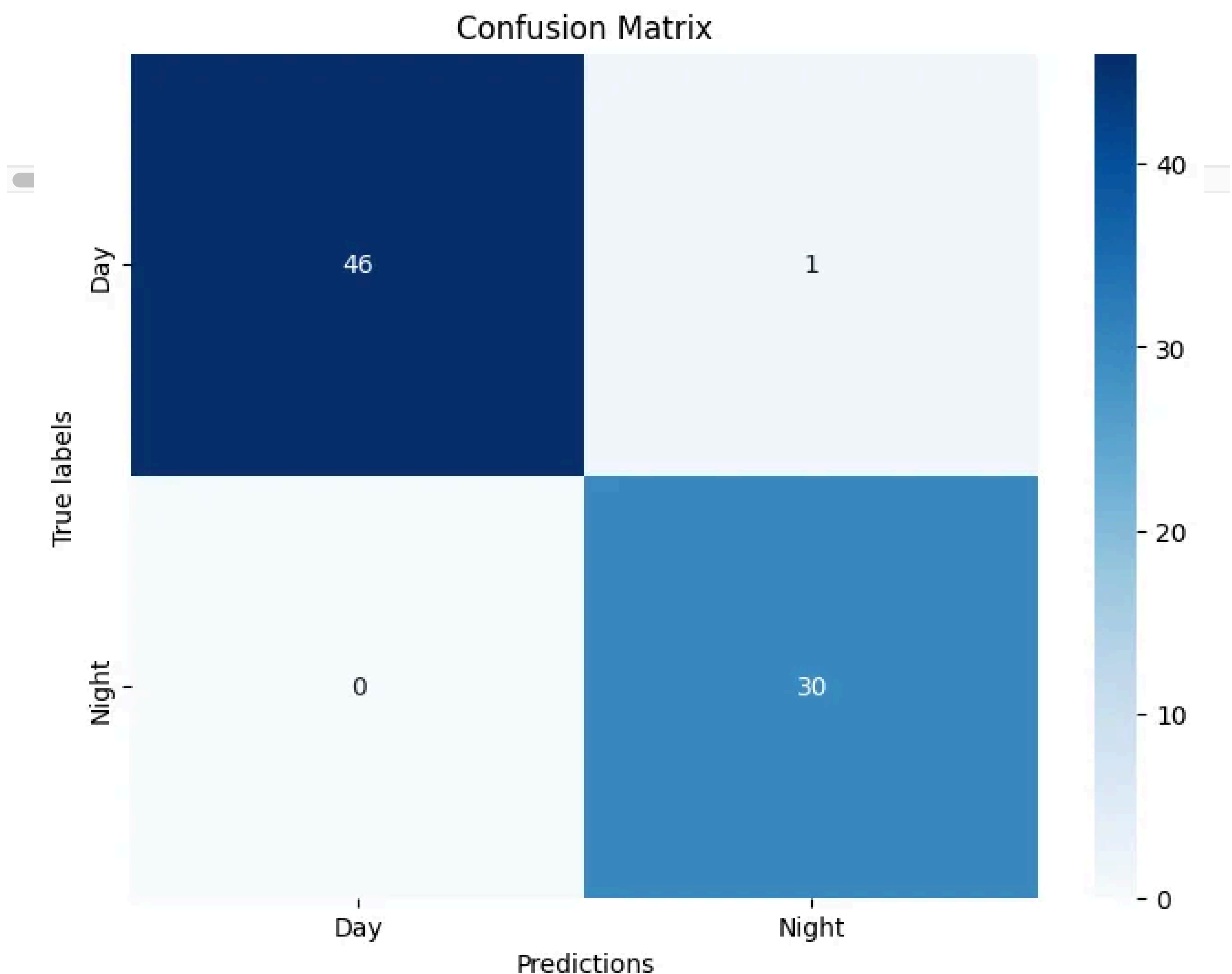


It seems to be effortless for the model :)

To verify if our model is good, we can use a confusion matrix:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Get true labels
y_true = validation_generator.classes
# Convert predicted probabilties in binary class
```

```
y_pred_binary = (pred_probs > 0.5).astype(int)
# Compute the confusion matrix
cm = confusion_matrix(y_true, y_pred_binary)
# Display the confusion matrix with seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, cmap="Blues", fmt="d", xticklabels=["Day", "Night"]
plt.xlabel("Predictions")
plt.ylabel("True labels")
plt.title("Confusion Matrix")
plt.show()
```



The better the model's performance, the darker the diagonal from the top-left to the bottom-right of the confusion matrix becomes. Here, we can clearly say it is a good result (maybe even too many neurons for this simple task?).

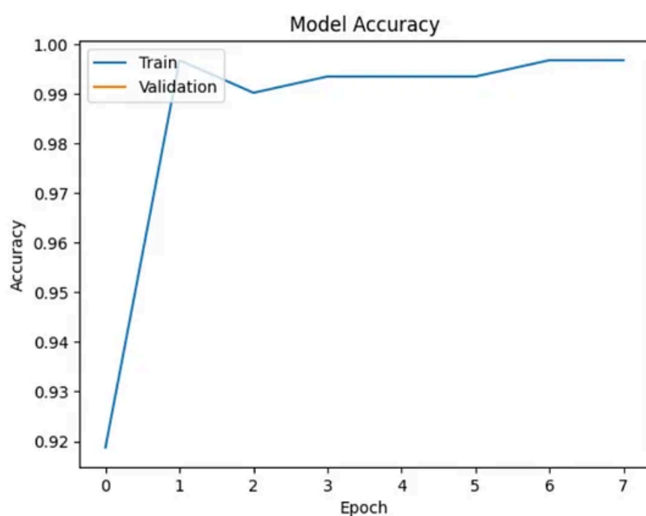
> *A* ***confusion matrix*** *is a* performance measurement *for classification algorithms that summarizes the number of correct and incorrect predictions made by the model on a dataset. It compares predicted classifications to actual classifications, organizing them into a matrix format where rows represent the true labels and columns represent the*

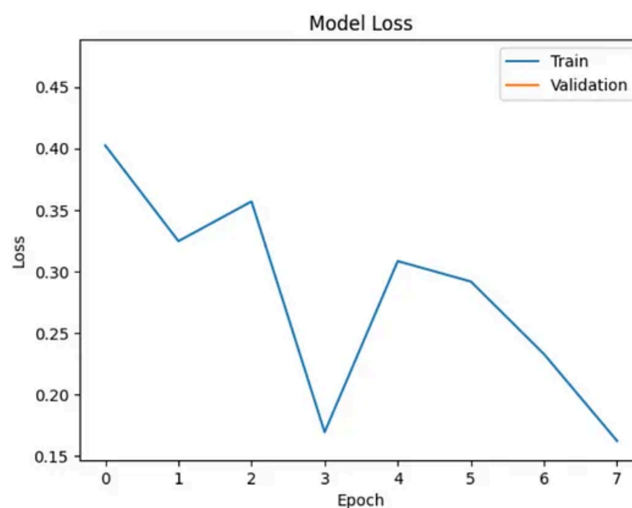
> *predicted labels. This allows for a quick visual inspection of how well the model is performing in terms of correctly and incorrectly classified instances across different classes. ([more info about it](#))*

Then we plot loss and accuracy to understand the effectiveness of our model:

```
val_loss, val_accuracy = model.evaluate(validation_generator, steps=20)

# Plot the training and validation accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
# Plot the training and validation loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
3/3 [==============================] - 10s 3s/step - loss: 0.0384 - accuracy: 0.9870
```





great accuracy but a weird loss curve

Since I have other tools in my bag, let’s see what else I can do :)

— — —

1.2 Building a Convolutional Neural Network (CNN) for Image Classification

The difference between CNN and DNN is that a CNN is a type of DNN with convolutional layers added before the others (flatten, dense, and output).

> *A **convolutional layer** applies* filters *to input data to extract features, enabling the neural network to learn spatial hierarchies and patterns within the data.*

The steps for making a convolutional neural network are the same. The only difference is that we now have two new types of special layers in our toolkit: the convolutional layer and the max pooling layer.

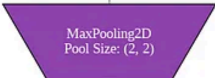
1.2.1 Building the Neural Network

```
# Initialize the model
model = Sequential()

# Add the first convolutional layer
model.add(Conv2D(32, (3,3), activation='relu', input_shape=input_shape))
# Add a max pooling layer
model.add(MaxPooling2D((2,2)))
# Add the second convolutional layer
model.add(Conv2D(64, (3, 3), activation='relu'))
# Add a max pooling layer
model.add(MaxPooling2D((2, 2)))
# Add the third convolutional layer
model.add(Conv2D(128, (3, 3), activation='relu'))
# Add a max pooling layer
model.add(MaxPooling2D((2, 2)))
# Flatten the output from the convolutional layers
model.add(Flatten())
# Add a fully connected layer
model.add(Dense(512, activation='relu'))
# Add the output layer
model.add(Dense(1, activation='sigmoid'))
# To display the CNN:
visualizer(model, file_name="model", file_format='png')
Image("model.png")
```

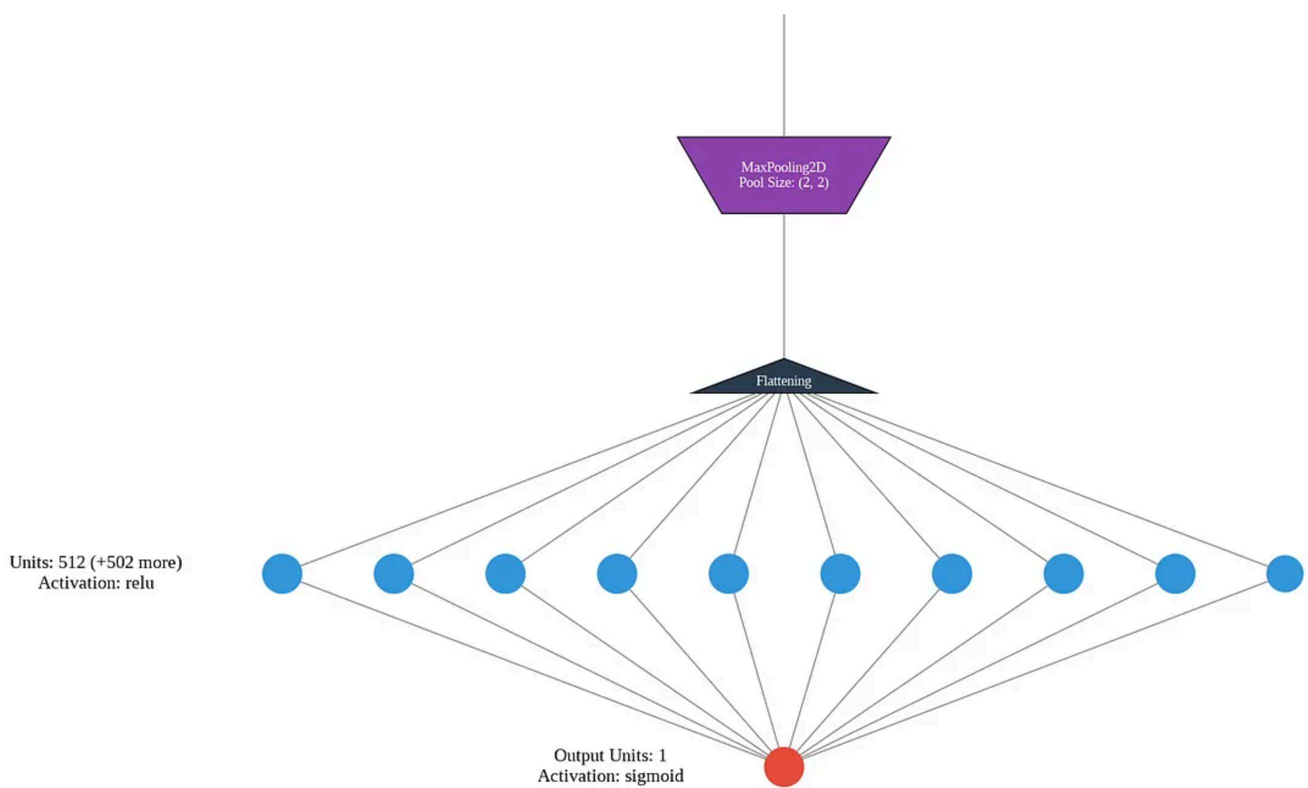
Image
150 x 110 pixels
RGB

Conv2D Layer
Kernel Size: (3, 3)
Filters: 32
Activation: relu
32
Feature Maps

MaxPooling2D
Pool Size: (2, 2)

Conv2D Layer
Kernel Size: (3, 3)
Filters: 64
Activation: relu
64
Feature Maps

MaxPooling2D
Pool Size: (2, 2)

Conv2D Layer
Kernel Size: (3, 3)
Filters: 128
Activation: relu
128
Feature Maps



You can see the layers here.

A convolutional layer applies filters to input data to extract features, while the max pooling layer reduces the spatial dimensions of the output volume, effectively downsampling and retaining the most important information. In our code snippet, `Conv2D(32, (3,3), activation='relu', input_shape=input_shape)` adds a convolutional layer with 32 filters of size 3x3, while `MaxPooling2D((2,2))` adds a max pooling layer with a pool size of 2x2.

But now the question is: Will there be more parameters to train or fewer parameters than for the fully connected network 1.1?

Let's run `model.summary()` to find out!

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_1 (Conv2D)           (None, 148, 108, 32)      896       
                                                                 
 max_pooling2d_1 (MaxPoolin  (None, 74, 54, 32)        0         
 g2D)                                                            
                                                                 
 conv2d_2 (Conv2D)           (None, 72, 52, 64)        18496     
                                                                 
 max_pooling2d_2 (MaxPoolin  (None, 36, 26, 64)        0         
 g2D)                                                            
                                                                 
 conv2d_3 (Conv2D)           (None, 34, 24, 128)       73856     
                                                                 
 max_pooling2d_3 (MaxPoolin  (None, 17, 12, 128)       0         
 g2D)                                                            
                                                                 
 flatten_1 (Flatten)         (None, 26112)             0         
                                                                 
 dense_2 (Dense)             (None, 512)               13369856  
                                                                 
 dense_3 (Dense)             (None, 1)                 513       
                                                                 
=================================================================
Total params: 13463617 (51.36 MB)
Trainable params: 13463617 (51.36 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This indicates we have around 13 million of parameters, which is half of what we had before! It can be explained by the number of filters we applied. Let's see if it performs better or not.

1.2.2 Training the Neural Network

Let's now compile the model with the same parameters as in Part 1:

```
# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'

# Train the model using the generator
history = model.fit(
    train_generator,
    steps_per_epoch=10,
    epochs=5,
    validation_data=validation_generator,
```

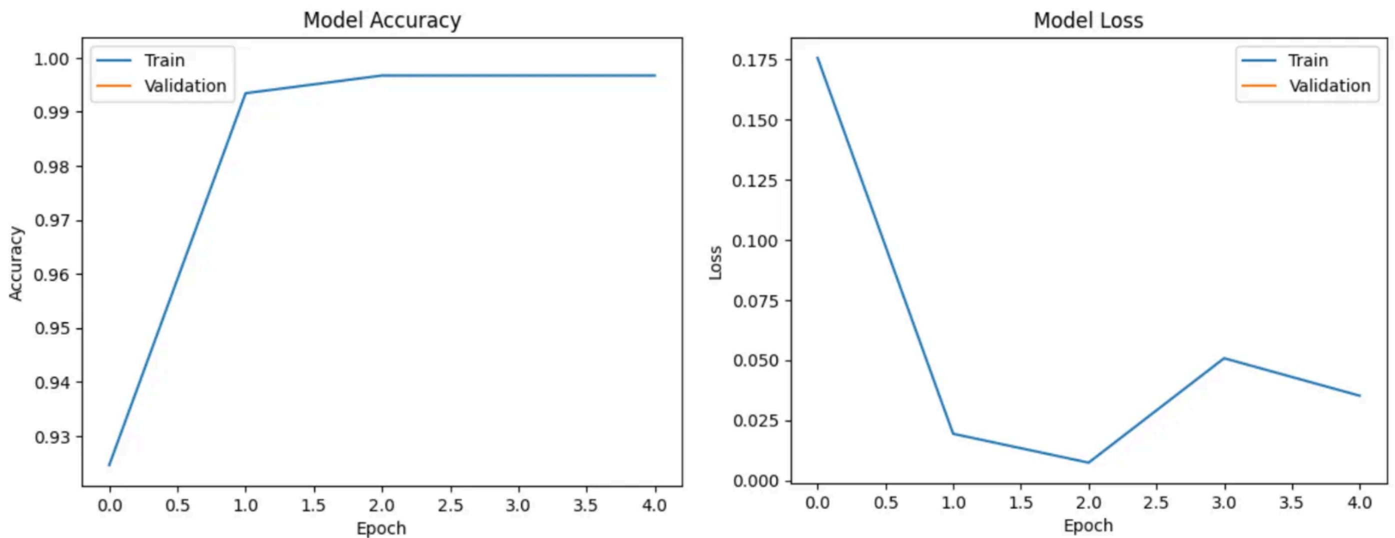
```
        validation_steps=20
    )
```

```
Epoch 1/5
10/10 [==============================] - ETA: 0s - loss: 0.1756 - accuracy: 0.9246WARNING:tensorflow:Your input ran out of data; interru
10/10 [==============================] - 64s 6s/step - loss: 0.1756 - accuracy: 0.9246 - val_loss: 0.0264 - val_accuracy: 1.0000
Epoch 2/5
10/10 [==============================] - 42s 4s/step - loss: 0.0194 - accuracy: 0.9934
Epoch 3/5
10/10 [==============================] - 44s 4s/step - loss: 0.0074 - accuracy: 0.9967
Epoch 4/5
10/10 [==============================] - 42s 4s/step - loss: 0.0508 - accuracy: 0.9967
Epoch 5/5
10/10 [==============================] - 42s 4s/step - loss: 0.0352 - accuracy: 0.9967
```

Remarquable high accuracy (0.99%)

1.2.3 Evaluate the model

This is the same as before, so here are only the results:



Results: good accuracy with a few epochs and a good-looking loss curve

Perfect predictions!!!

Because we already have a perfect score, let's continue with the classification of land/ocean, and we will discover a new use of DNN.

Part 2: Land or ocean?

Now comes the challenging part, even a human has difficulties distinguishing these types!

2.1 Data preparation and importation

For this, I unified my previous folder with other pictures from the same camera available on Flickr. I sorted the images between land and ocean for quite a time to make my folder with the following structure:

```
.
└── dataset/
```

```
    ├── land/
    │   └── *298 pictures*
    └── ocean/
        └── *314 pictures*
```

However, to use the previous models, I need to modify the structure in order to have the same base as before:

```
# Load dataset:
!gdown 1jy5b4TCGzcYxa-pIBKQZDG20n6rPHYGU && unzip -qq landOrOcean.zip

import shutil
# Define the source and destination directories
source_dir = 'dataset'
destination_dir = 'landoroceancnn'
# Check if the source directory exists
if not os.path.exists(source_dir):
    raise Exception(f"Source directory '{source_dir}' does not exist.")
# Check if the destination directory already exists
if os.path.exists(destination_dir):
    # If it exists, ask the user if they want to overwrite it
    overwrite = input(f"Destination directory '{destination_dir}' already exist
    if overwrite.lower() != 'y':
        print("Copying canceled.")
        exit(1)
# Copy the source directory recursively to the destination directory
shutil.copytree(source_dir, destination_dir)
print(f"Successfully copied '{source_dir}' to '{destination_dir}'.")
# Select the directory where the data is saved
data_dir = os.path.join('landoroceancnn')  # Assuming dataset is in the current
# Define function to extract class label from folder name
def get_label(folder_path):
  return folder_path.split("/")[-1]  # Extract class from last folder name
# Create separate training and validation splits manually
train_data_dir = os.path.join(data_dir, 'train')
validation_data_dir = os.path.join(data_dir, 'validation')
# Create the train and validation directories if they don't exist
if not os.path.exists(train_data_dir):
  os.makedirs(train_data_dir)
if not os.path.exists(validation_data_dir):
  os.makedirs(validation_data_dir)
# Move training and validation data to respective directories
# (Assuming you have the 'land' and 'ocean' folders within the 'landoroceancnn'
for class_dir in ['land', 'ocean']:
  src_dir = os.path.join(data_dir, class_dir)
  for filename in os.listdir(src_dir):
    file_path = os.path.join(src_dir, filename)
    target_dir = os.path.join(train_data_dir if random.random() < 0.8 else vali
```

```
        os.makedirs(target_dir, exist_ok=True)
        shutil.move(file_path, target_dir)
```

Thanks to ChatGPT, who helped me a little bit to write this code, I now have a folder like this:

```
.
└── landoroceancnn/
    ├── train/
    │   ├── land/
    │   └── ocean/
    └── validation/
        ├── land/
        └── ocean/
```

```
[5] # Select the directory where the data is saved
    data_dir = 'landoroceancnn/'

    # Set the path to the directory where your data is stored
    train_directory = os.path.join(data_dir, 'train')
    validation_directory = os.path.join(data_dir, 'validation')

    # Setting the data directories for each class in the train set
    train_day_dir = os.path.join(data_dir, 'train/land')
    train_night_dir = os.path.join(data_dir, 'train/ocean')

    # Setting the data directories for each class in the train set
    validation_day_dir = os.path.join(data_dir, 'validation/land')
    validation_night_dir = os.path.join(data_dir, 'validation/ocean')
```

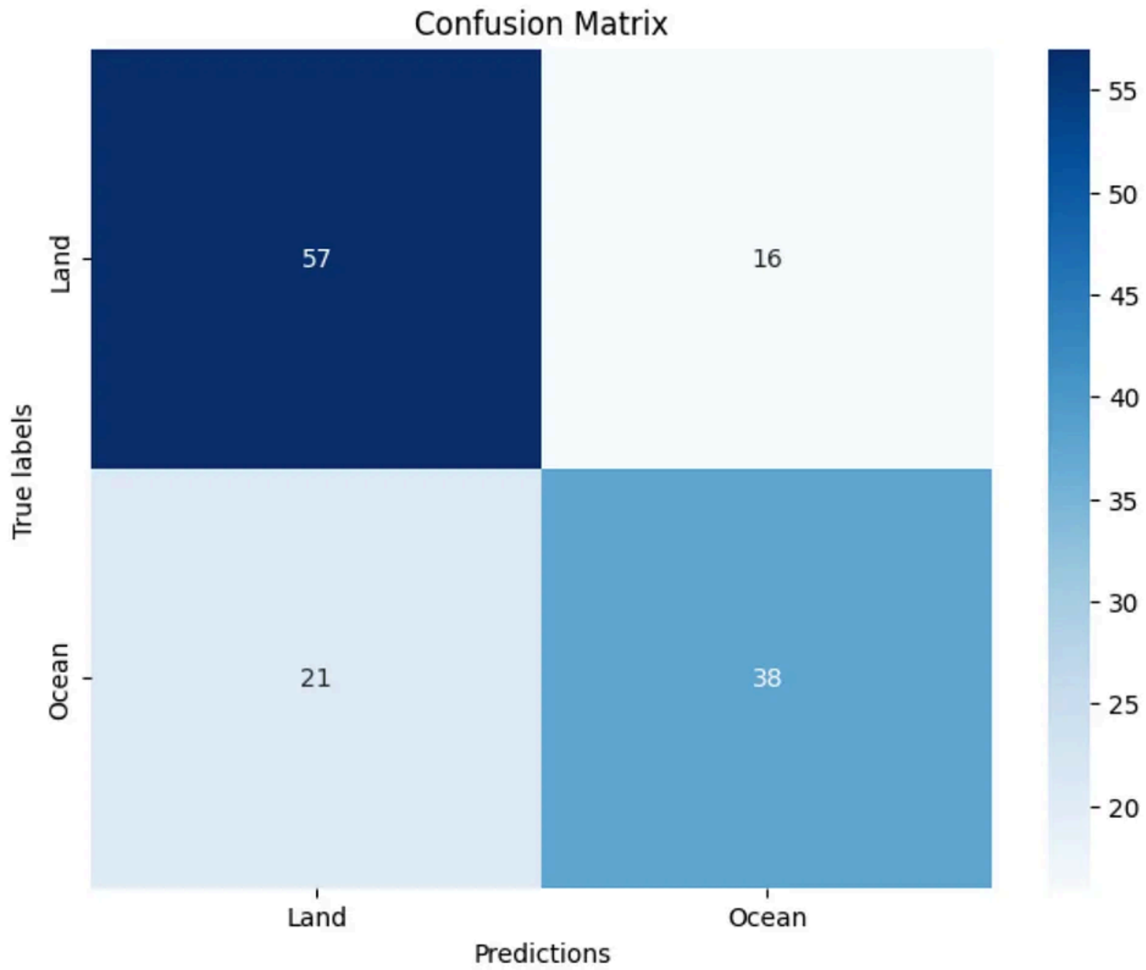
```
[6]
    # Load the training data using the ImageDataGenerator class
    train_datagen = ImageDataGenerator(rescale=1./255)
    train_generator = train_datagen.flow_from_directory(
        train_directory,
        target_size=(150, 110),
        batch_size=32,
        class_mode='binary'
    )

    Found 480 images belonging to 2 classes.
```

```
    # Load the validation data using the ImageDataGenerator class
    validation_datagen = ImageDataGenerator(rescale=1./255)
    validation_generator = validation_datagen.flow_from_directory(
        validation_directory,
        target_size=(150, 110),
        batch_size=32,
        shuffle = False,
        class_mode='binary'
    )

    Found 132 images belonging to 2 classes.
```

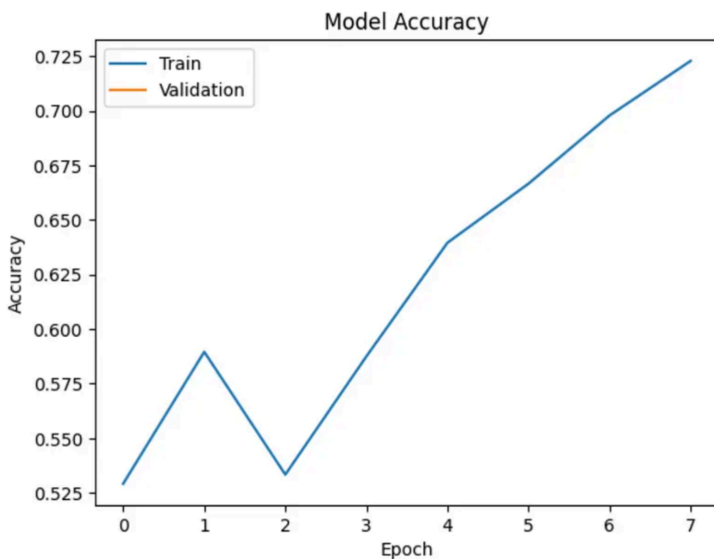
Screenshots of my code in the Colab Notebook to import and split data

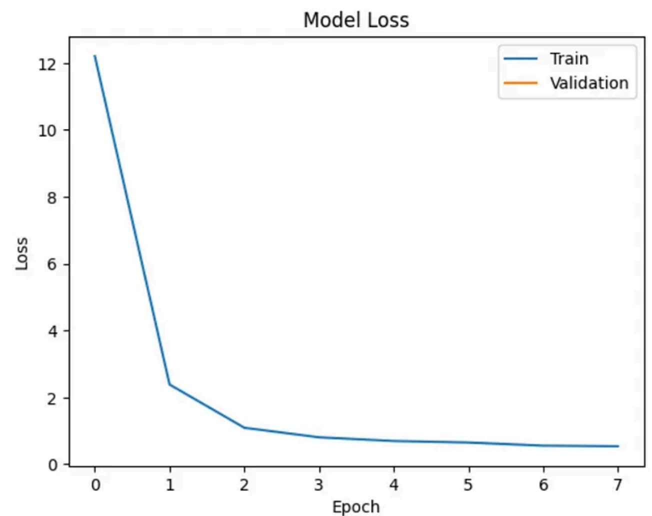
2.2 First DNN:

Now we can train our first DNN and evaluate it:



The final accuracy remains at 0.7, which is quite good for AI, but there is room for improvement.

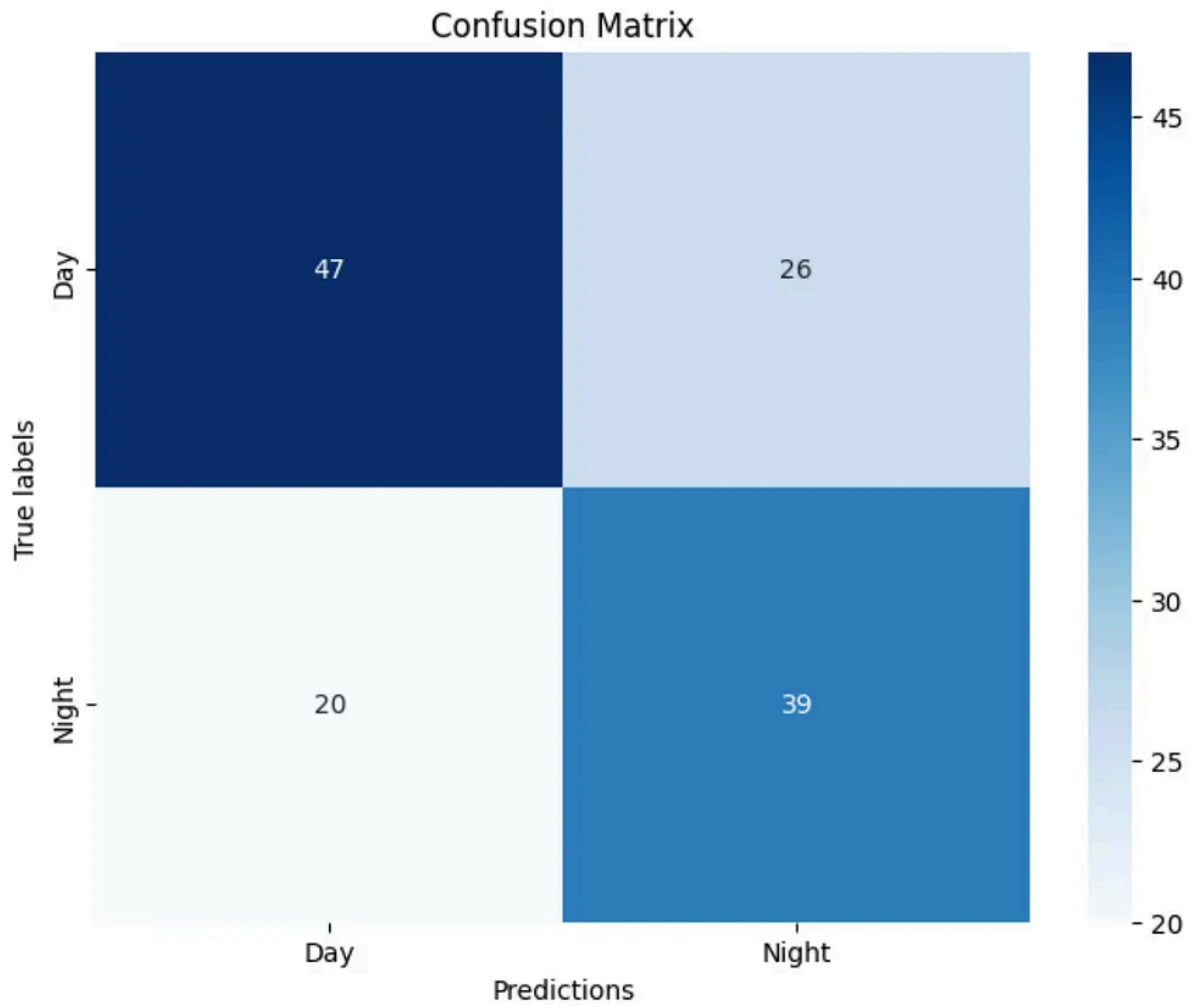




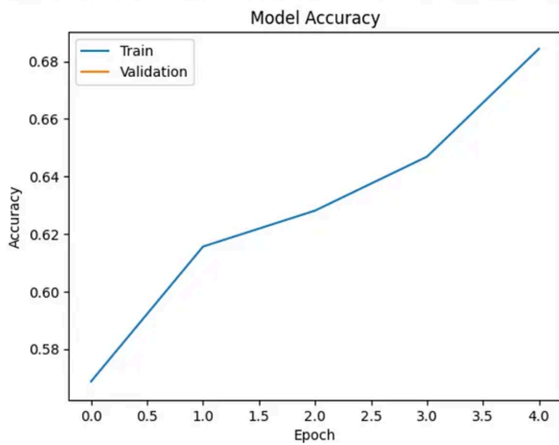
The model's loss appears to stagnate after the 4th epoch, while its accuracy continues to improve. This suggests that additional data may be needed to achieve better results with the current model.

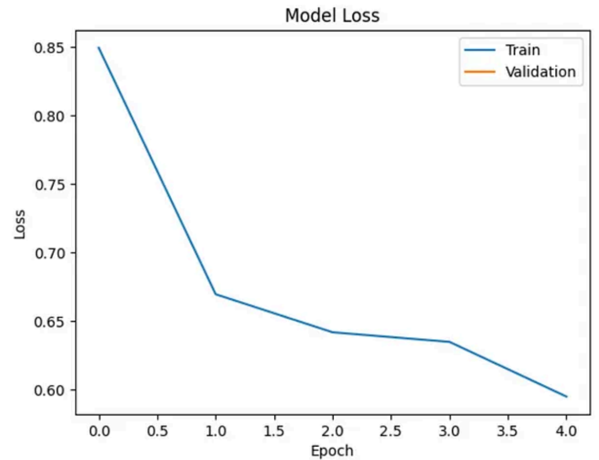
2.3 With the previous CNN

Results:




```
5/20 [======>.......................] - ETA: 1:04 - loss: 0.6403 - accuracy: 0.6515WARNIN
20/20 [==============================] - 21s 912ms/step - loss: 0.6403 - accuracy: 0.6515
```





Not really better than the previous DNN maybe it has too many filters that hide information, or not enough epochs, but because we don't want to wait for more epochs, let's try a new way:

2.4 Using a Pre-Trained Model: land or ocean?

Let's introduce a new use of CNN/DNN: **pre-trained models.**

One way to improve our model would be to train it on a much larger dataset. Unfortunately, we don't have access to more data for now. In this part we will use **transfer learning,** which means we will use a model that has been previously trained on a large dataset and modify it for our needs.

Today, we will use the VGG16 model, which is a CNN with 16 layers that has been trained on the ImageNet dataset. We will use this model to classify images into two categories: 'land' or 'ocean'.

2.4.1 Splitting dataset

From the "dataset" folder:

```
# Remember that the batch size is the number of images that will be processed
# the model is updated. One epoch is one full pass through the dataset.
BATCH_SIZE = 32

# Size of the images. If the input images are a different size, they will be
# resized to these dimensions.
IMG_SIZE = (150, 110)
# Directory the data is saved.
directory = "dataset/"
# We are using keras's image_dataset_from_directory function to load the images
# from the directory. When we use this function, we can tell keras if it is a
# validation or training set. Just make sure to use the same seed for both to
# make sure there is no overlap in the datasets.
train_dataset = image_dataset_from_directory(directory,
                                             shuffle=True,
                                             batch_size=BATCH_SIZE,
                                             image_size=IMG_SIZE,
                                             validation_split=0.2,
                                             subset='training',
                                             seed=42)
validation_dataset = image_dataset_from_directory(directory,
                                             shuffle=True,
                                             batch_size=BATCH_SIZE,
                                             image_size=IMG_SIZE,
                                             validation_split=0.2,
                                             subset='validation',
                                             seed=42)
```

2.4.2 Building a new neural network using a pre-trained model

Data augmentation techniques like `RandomFlip` and `RandomRotation` create variations in the training data. RandomFlip horizontally or vertically mirrors the image, while

RandomRotation twists it by a random angle. This helps the model recognize objects regardless of orientation or viewpoint, making it more robust.

Since VGG16 was trained on a vast dataset containing various objects and scenes, it has already learned valuable features for image recognition. We can leverage this knowledge by:

- **Freezing Pre-trained Layers:** We'll freeze the weights of most layers in VGG16. This prevents them from being re-trained during our land/ocean classification task. By freezing these layers, we essentially "lock in" the general image recognition capabilities learned from ImageNet.
- **Training Final Layers:** We'll add new final layers on top of the pre-trained VGG16 model. These final layers will be specifically trained to distinguish between land and ocean images. This approach allows us to focus our training effort on the task at hand while benefiting from the extensive knowledge of the pre-trained model.

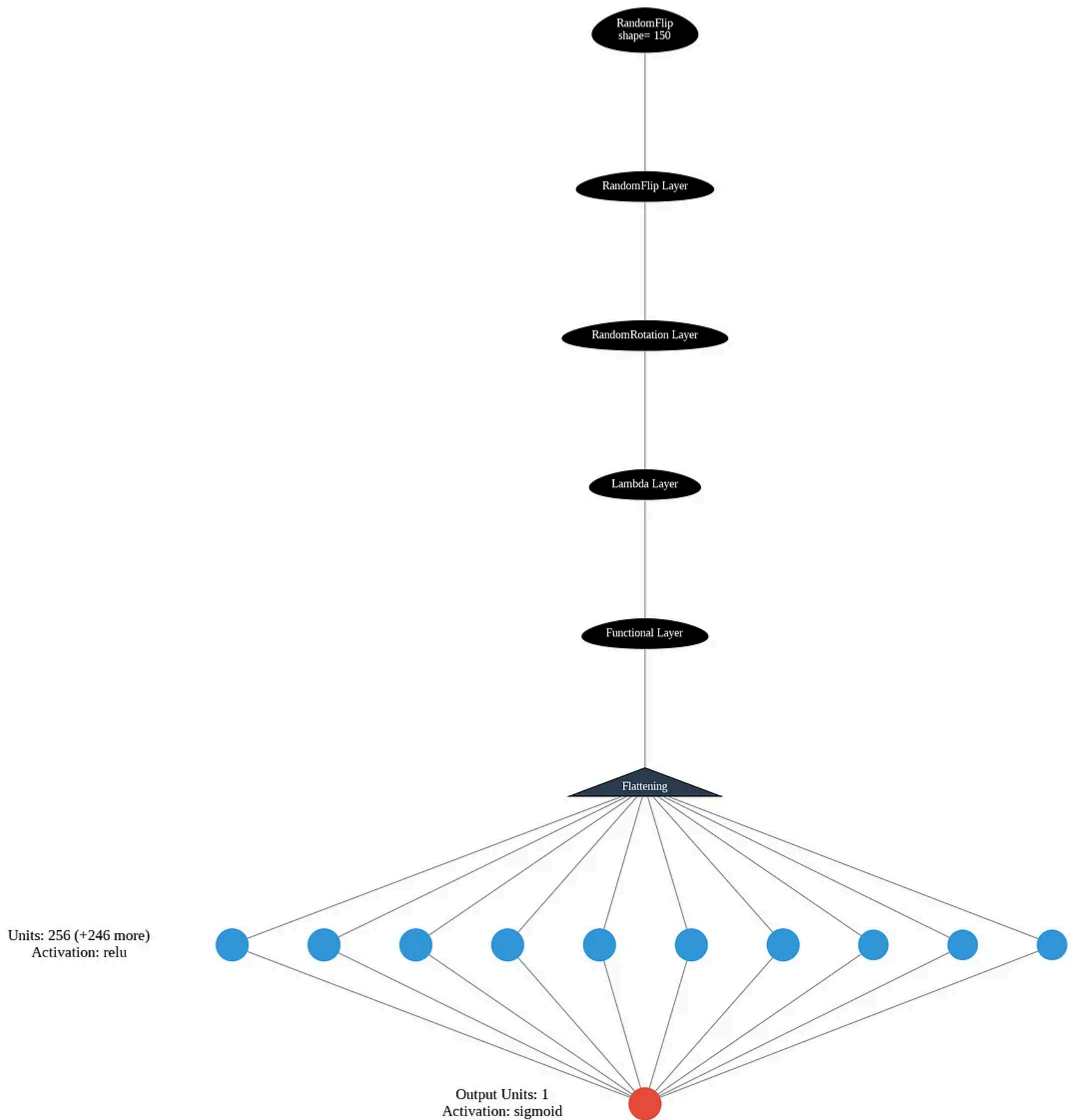
```
# Define the input shape for the model
input_shape = (150, 110, 3)

# Initialize the model
new_model = Sequential()
# Add input layer
new_model.add(Input(shape=input_shape))
# Add data augmentation layers
new_model.add(RandomFlip('horizontal'))
new_model.add(RandomRotation(0.2))
# You do not need to worry about this step. But in case you are curious:
# The "Lambda" layer is another special layer that let's you apply a custom
# function to the data. Because we know we're going to use VGG16, we will use
# the same pre-processing function that was used when it was first trained.
new_model.add(
    Lambda(
        preprocess_input,
        name='preprocessing',
        input_shape=input_shape
    )
)
# Load a pre-trained model
VGG = VGG16(
    # we want to use our own final layers customized for our task
    include_top=False,
    input_shape=input_shape,
    weights='imagenet'
```

```
    )
# Freeze parameters each layer because we only want to train the final layers
for layer in VGG.layers:
  layer.trainable = False
# Add the pre-trained model to our model
new_model.add(VGG)
# Add a flattening layer
new_model.add(Flatten())
# Add a fully connected layer
new_model.add(Dense(256, activation='relu'))
# Add the output layer
new_model.add(Dense(1, activation='sigmoid'))
new_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights
58889256/58889256 [==============================] - 2s 0us/step
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 150, 110, 3)       0

 random_rotation (RandomRot  (None, 150, 110, 3)       0
 ation)

 preprocessing (Lambda)      (None, 150, 110, 3)       0

 vgg16 (Functional)          (None, 4, 3, 512)         14714688

 flatten_2 (Flatten)         (None, 6144)              0

 dense_5 (Dense)             (None, 256)               1573120

 dense_6 (Dense)             (None, 1)                 257

=================================================================
Total params: 16288065 (62.13 MB)
Trainable params: 1573377 (6.00 MB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Note: less than 2 millions of trainable parameters for the same amount of our CNN's parameters ! Will it be better ?

Let's take a loot at the new model with `visualizer()` :



Not so many new neurons :)

2.4.3 Training and Evaluation

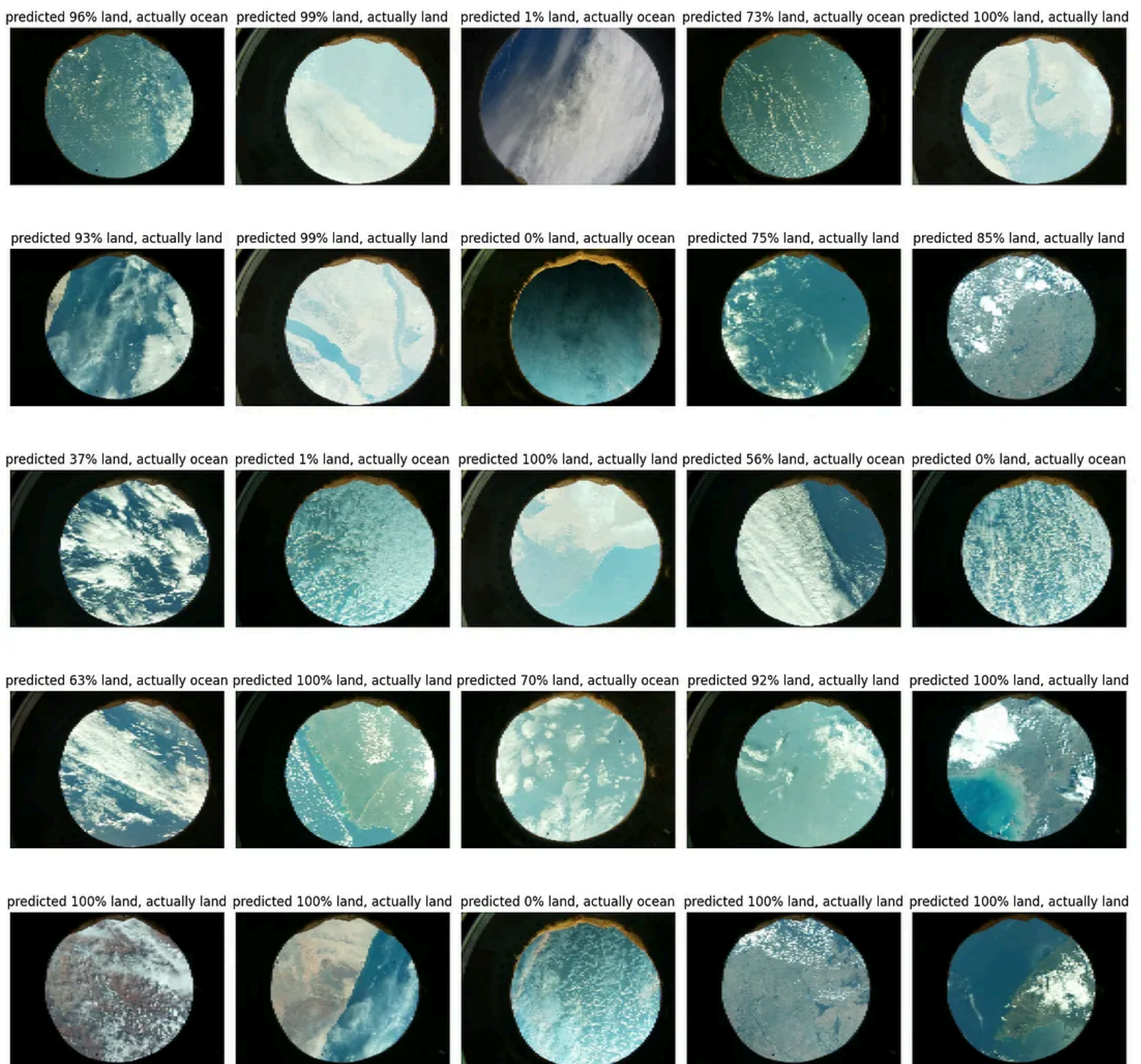
The code is similar, so I will only show you the results.

> *A **batch** is a subset of the dataset used for one update of the model's parameters.*
>
> *An **epoch** is a single pass through the entire dataset during training.*
>
> *A **step**, also known as an iteration, is one update of the model's parameters using one batch of data.*

```
Epoch 1/10
16/16 [==============================] - 184s 10s/step - loss: 6.2832 - accuracy: 0.6633 - val_loss: 2.3865 - val_accuracy: 0.7295
Epoch 2/10
16/16 [==============================] - 173s 9s/step - loss: 1.5804 - accuracy: 0.7837 - val_loss: 0.9537 - val_accuracy: 0.8115
Epoch 3/10
16/16 [==============================] - 186s 10s/step - loss: 0.9416 - accuracy: 0.8041 - val_loss: 0.7381 - val_accuracy: 0.8115
Epoch 4/10
16/16 [==============================] - 178s 9s/step - loss: 0.8110 - accuracy: 0.8000 - val_loss: 0.6682 - val_accuracy: 0.8115
Epoch 5/10
16/16 [==============================] - 171s 9s/step - loss: 0.5443 - accuracy: 0.7939 - val_loss: 0.3831 - val_accuracy: 0.8197
Epoch 6/10
16/16 [==============================] - 186s 10s/step - loss: 0.2816 - accuracy: 0.8878 - val_loss: 0.4445 - val_accuracy: 0.8033
Epoch 7/10
16/16 [==============================] - 182s 10s/step - loss: 0.2428 - accuracy: 0.9000 - val_loss: 0.6816 - val_accuracy: 0.8115
Epoch 8/10
16/16 [==============================] - 177s 9s/step - loss: 0.3044 - accuracy: 0.8735 - val_loss: 0.5906 - val_accuracy: 0.8279
Epoch 9/10
16/16 [==============================] - 181s 10s/step - loss: 0.2824 - accuracy: 0.8898 - val_loss: 0.4712 - val_accuracy: 0.8443
Epoch 10/10
16/16 [==============================] - 178s 10s/step - loss: 0.2347 - accuracy: 0.9204 - val_loss: 0.5813 - val_accuracy: 0.8197
```

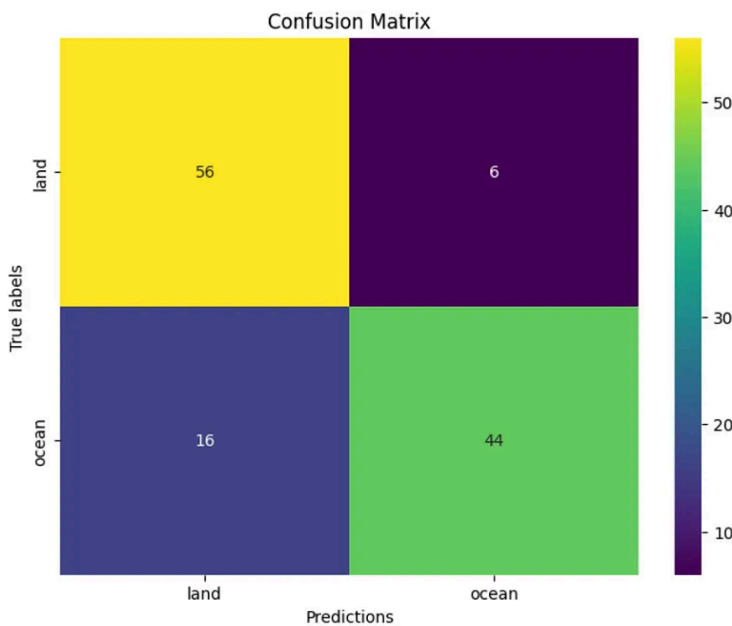
It seems that we achieve an accuracy of around 0.9, way better than the other models!

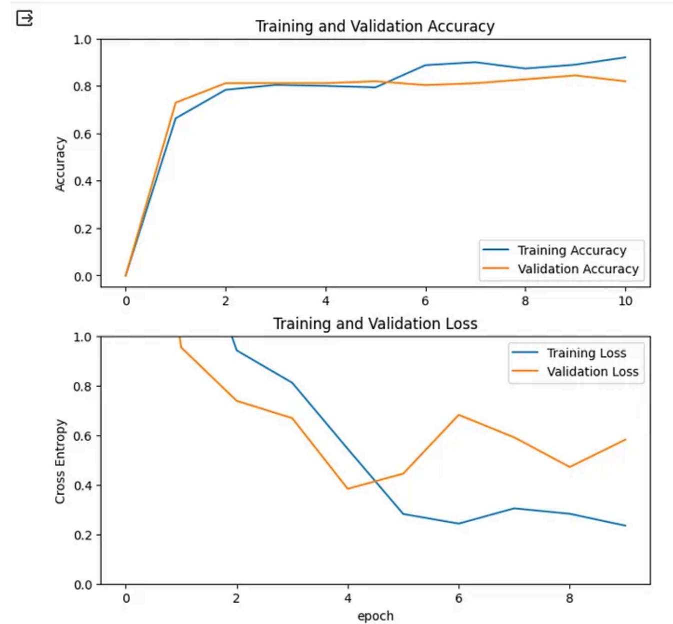
Here is an excerpt of the validation data:



The model looks unerring :)

We can display a confusion matrix and some graphs to verify our hypothesis:





The model indeed did great!

So far, our best algorithm is the pre-trained model ! So we have an absolute winner for this challenge!

Conclusion

In this project, I explored three machine learning models (DNN, CNN, and a pre-trained model) to classify images from the International Space Station (ISS) camera. While all three models achieved high accuracy in distinguishing day from night (a relatively simple task), the pre-trained model emerged as particularly efficient. This efficiency stems from its significantly lower number of trainable parameters, making it ideal for resource-constrained devices like Raspberry Pi.

Next, I challenged the models with a more complex task: differentiating between land and ocean. The DNN struggled, potentially due to a limited number of neurons or layers. Conversely, the pre-trained model excelled, demonstrating its adaptability.

Model Type	Trainable Parameters (Approx.)	Performance (Day/Night)	Performance (Land/Ocean)	Advantages	Disadvantages
DNN (Deep Neural Network)	High (Varies by architecture)	High	May struggle (Limited feature extraction)	Flexible architecture	High training requirements, may need more data
CNN (Convolutional Neural Network)	Moderate	High	Good	Excellent for image data, learns spatial features	Requires careful hyperparameter tuning
Pre-trained Model	Low (Already trained on large datasets)	High	High	Efficient, requires less data for fine-tuning	Less flexible architecture, may require transfer learning techniques

Summary of Image Classification Models

Moving forward, I plan to deploy these models on my Raspberry Pi to automatically categorize daytime photos as land or ocean during future space photography endeavors. Land images offer a more visually engaging experience compared to vast stretches of ocean. I would use a pre-trained model as it costs less memory while being super efficient, so I could save more pictures on the device!

For future exploration, I'm intrigued by the possibility of training a model to recognize different cloud types. By feeding the model pairs of images with a slight time delay, mimicking human 3D vision, we could achieve cloud classification. However, this would require a labeled dataset with precise cloud identification, potentially a more challenging hurdle than image sorting for land/ocean distinction.

3D picture of Crete (Greece) — Several techniques allow to see images in 3d. Here are a few. Using a virtual reality headset; squinting while looking at the right and left images until you mentally superimpose them; displaying the images behind Fresnel lenses.

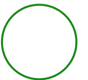
This project has been a rewarding experience. Not only did I leverage my newly acquired AI knowledge from TCS, but I also utilized a custom dataset I created through meticulous sorting of images — a time-consuming task that ultimately yielded impressive results.

This article was developed as a final project in the *Introduction to Artificial Intelligence* course offered by The Coding School (TCS) and supported by the Department of Defense STEM.

About The Coding School: TCS is a 501(c)(3) nonprofit focused on skill building and workforce development in technologies that are going to change the world over the next decade, such as Quantum Computing and Artificial Intelligence. With a commitment to accessible, supportive computer science education, TCS offers first-of-their-kind pathways and programs for individuals at every stage of their STEM journey, including: K-12 students and educators, community college and university

students and faculty, and members of the workforce. Since 2014, TCS has trained over 50,000 individuals from 125 countries in partnership with leading companies and universities such as MIT, Google, and IBM. For more information about TCS and how to get involved or support its initiatives, please visit the-cs.org.

The Coding School | Artificial Neural Network | Artificial Intelligence | Internationalspacestation | Image Classification



Follow

Written by Eva

0 Followers

Sometimes I learn about AI :)

Recommended from Medium