Udacity Online School of Artificial Intelligence
Machine Learning Engineer Nanodegree
Adriano Rodrigues Figueiredo Torres

Capstone project

Application of Convolutional Neural Networks in Landscape Monitoring

Problem Context and Definition

Overview

Computer vision is rapidly gaining traction as an emergent technology. Its recent applications range from simpler tasks, such as recognizing characters on a screen, to more elaborate ones, such as equipping autonomous cars with means of sensing their surrounding environment.
Amongst the vast array of possibilities where image recognition can be applied is the development of tools which can help us preserve large natural vegetation regions. In this type of scenario, one could apply computer vision techniques to monitor how landscapes change over time. Such information could assist the creation of models for measuring human footprint. Furthermore, such monitoring systems could be applied in the prevention of illegal devastation and exploitation of land, or even to help discover new species.
In this project, we are going to develop a basic image recognition system whose goal is to accurately predict whether there is a cactus species in the input images. The idea could then be extended to the numerous species that compose a biome, providing a useful tool for monitoring changes in such ecosystems.

Problem Statement

The goal of this project consists in, given an input dataset of thousands of landscape images that have been previously labeled, developing an algorithm capable of accurately predicting if specimens of the Neobuxbaumia tetetzo cactus species are present. The input dataset is available at a playground competition in Kaggle. The original version of the dataset was generated by the VIGIA project, an initiative of researchers in Mexico whose project is to assess and measure the impact of human activity in climate change, and how thesy reflect on the planet's fauna and flora. The main steps to be followed consist of:

1 – Downloading Kaggle's version of the dataset
2 – Performimg exploratory analysis of the input dataset
3 – Developing a neural network model to classify the images
4 – Training the model against a training set
5 – Running the model in the test set
6 – Benchmarking the model against the established metrics

Our solution will consist of a Convolutional Neural Network (CNN) model, which is an example of a deep learning algorithm. Deep learning methods comprise a set of algorithms that attempt to build high-level abstractions of data, by cascading layers of non-linear processing units. CNNs have been successful in image recognition tasks, which motivates us to apply them to solve our task.
We expect the final model to show high accuracy (as defined in the Metrics section below), so as to establish a baseline architecture for extending it to detect other species.

Metrics

If we were to submit our model to Kaggle, it would be tested against the Receiver Operating Characteristics curve. The method consists in plotting the rate of true positives, which we are going to call sensitivity, against the rate of false positives, or the probability of fall-out.
In a binary classification problem, for example, there are four possible scenarios:

1 – The model predicts there is a cactus in the picture and the cactus is actually in the picture – a true positive (TP);
2 – The model predicts there is a cactus in the picture, when in fact there is no cactus – a false positive (FP);
3 – The model predicts there is no cactus in the picture, but there exists a cactus in the picture – a false negative (FN)
4 – The model predicts there is no cactus in the picture, and indeed there is no cactus – a true negative (TN).

We define the true positive rate as the quotient between true positives and all the positive predictions:

$$TPR = TP/(TP+FN)$$

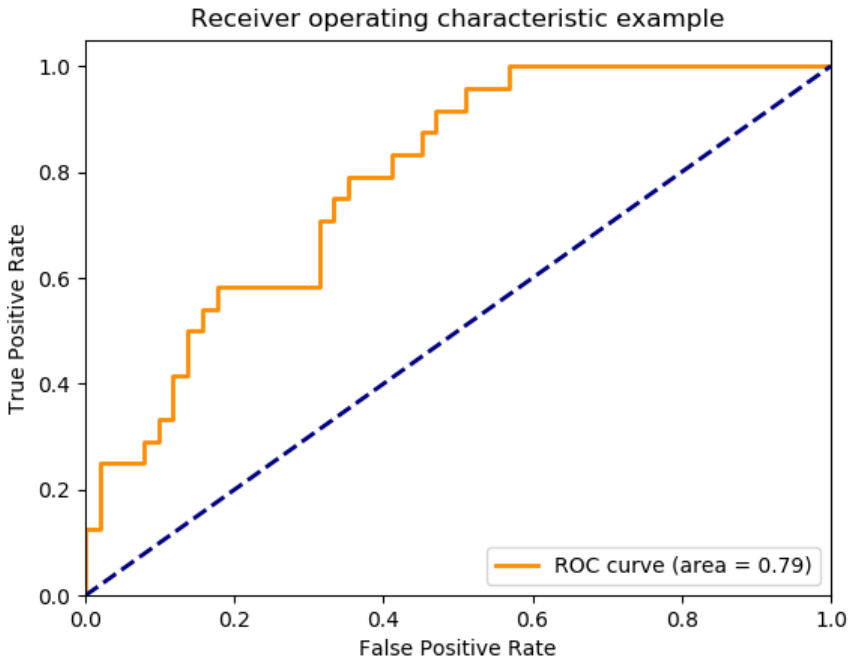
Analogously, the false positive rate consists of all the false positives divided by all the negative values:

$$FPR = FP/(FP+TN)$$

We define accuracy as:

$$ACCURACY = (TP+TN)/(TP+TN+FP+FN)$$

The following picture, extracted from the scikit-learn documentation, depicts an ROC curve:



The diagonal represents a random guess. We thus want our model to perform well above the diagonal. The closest the ROC curve area gets to 1, the better our model is at predicting the target feature.

For our learning purposes, we will adopt accuracy as the main metric.

Analysis

Data Exploration

The dataset is available in the Data section of the Aerial Cactus Identification playground competition. The picture have been resized by Kaggle so as to generate a uniform dataset of 32 x 32 thumbnail images in the JPEG format. There are two columns in the dataset:

1 – id: A string of alphanumeric digits
2 – has_cactus: A column to indicate the presence (represented by 1), or the absence (represented by 0) of a cactus in the image.

Two folders are present in the dataset:

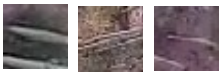
1 – train: a sample of 17500 pictures which shall be used for training the model
2 – test: a folder which contains 4000 pictures, which we will use to run our classifier on and measure its performance.

Finally, there is the train.csv file in the root folder. It contains entries for all the training pictures. We assume all the data to be correctly labeled by this file.

Image samples

The first three images in have been randomly chosen from the train folder, whilst the last three were picked from the test folder.

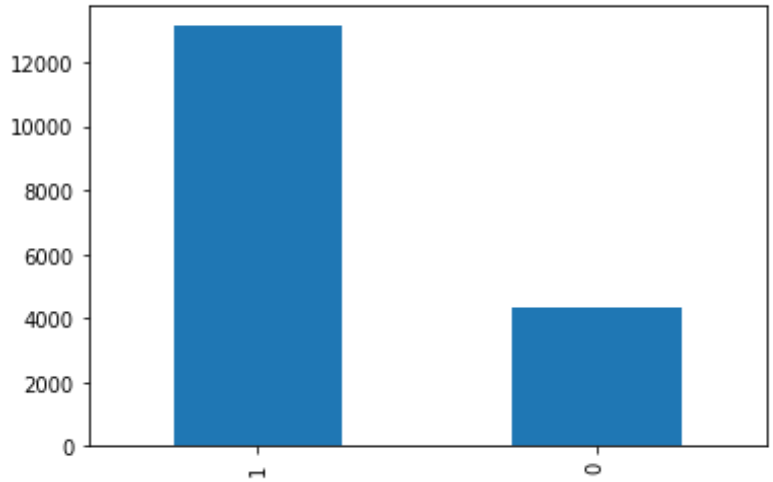
Random samples from training set



Random samples from test set

As expected, we cannot draw much information to make sense of the images, since they are only 32 x 32 pixels, and zooming in would degrade quality.

In the training set, there are 13136 images where has_cactus = 1, and 4364 images where has_cactus = 0, as shown by the following histogram (for code on how to generate the plot and count the values, please refer to the Jupyter Notebook file, which contains all the code we will use for this project).



Histogram of the training set according to has_cactus values

Algorithms and Techniques

The classifier we are going to use is the Convolutional Neural Network, or CNN. Our motivation for using this model is based on several cases of it being successfully employed in several image recognition problems. The initial inspiration for the creation of such networks was in fact drawn from the animal visual cortex. Combing gradient descent steps and back-propagation to automate the process of learning, Yann LeCun et al. [1] devised an approach which became foundational in computer vision.
This method can be applied, but it is not restricted to, binary classification problems, which is our case. In fact, it can be extended to multiple classification problems. The model outputs a value corresponding to the probability of an item belonging to one or more classes. So, for example, we could use CNNs to attempt to distinguish between cats and dogs in a set of images, a type of binary classification problem. Moreover, we could write a model that attempts to differentiate between different breeds of dogs. In the latter case, we would be dealing with a multiple classification problem.
To enhance our model's performance, we can make tweaks to how to some of the parameters we pass to our program. This is commonly called hyperparameter tuning. In CNNs, we can tune some quantities:

Learning rate: Commonly represented by the greek letter alpha, this quantity basically means how fast (or slow) the weights are going to be shifted at each gradient descent step. We can set a fixed learning rate and keep it throughout the entire execution, but we can also dynamically update it as we progress in the learning process;

Number of epochs: An epoch is defined by an entire pass through the entire training set. In simple terms, we increase the number of epochs so as to minimize the gap between the test error and the training error;

Batch size: Refers to how many samples (images, in our case) are going to be processed by the network at each training step;

Activation function: In simple terms, an activation function is a function that is applied to the output of a neuron in the network. In a sense, this controls the degree of non-linearity of our network;

Number of hidden layers and units: In a neural network, we can define a layer as a set of neurons that are at the same level. As we add more layers, we tend to reduce the error of the network. However, the computational complexity of the model escalates, requiring more time and memory for the algorithm to run. We generally gradually add layers until the test error stops improving;

Weigh initialization: The way by which we assign the initial weights for each edge of the network. The way we initialize them can impact the model's performance.

Dropout: The ability to choose whether to deactivate a neuron during a test episode. This is usually done to avoid overfitting.

Benchmarking

To create a benchmark model for our problem, we are going to fit a simplistic network, meaning one with a very low number of epochs (1), and a small batch size of 8. This network is only going to contain a single convolutional layer. These are, in a way, unrealistic values. Under normal circumstances, we would not even begin training with such slow values; our starting values would naturally be higher than these. However, for educational purposes, and to see how much more accurate the model gets as we tune the parameters better, we are going to create this fictitious initial model. We will then tune it and (hopefully) see how much better the performance becomes.

Methodology

Generally, when dealing with data, we must initially perform some time of preprocessing, whether to convert it to a format our models can operate on, or to make the samples more uniform. Sometimes we also do it to remove outliers, deal with missing values, or to simply correct discrepancies that might have been introduced when the dataset was created. Since we are dealing with a didactic dataset, Kaggle was kind enough to do this step for us, and all the data is in the 32x32x3 format. Thus we will not need to deal with it here.

As a second step, we are going to create the benchmark model, and train it on a random subset (7500 entries) of the training dataset. We will print its performance and save the model.

Finally, we are going to gradually add more layers to the model, and tune the batch size and number of epochs. After we are satisfied with the result (more than 99% accuracy), we will save the final model, print its summary, and calculate its ROC score. We will then run the model on the test set.

Implementation

We will try to keep our implementation as compact and simple as possible. To have a more reusable code, we are going to define some functions:

- load_images: takes the path of a folder and loads the images of that folder into memory
- build_model: takes as input the number of levels of convolutional layers, adds these layers, then flattens them, and creates the output layer

- add_layer: an auxiliary function that gets called by the former. Using this approach, we are going to be able to reuse the implementations for both the benchmark and the improved models.

The following snippets detail the code for each function.

```
def load_images(path):
    images = {}
    for file in os.listdir(path):
        filename = os.path.join(path,
file)
        images[file] =
cv2.imread(filename)
    return images

def add_layer(model, **args):
    model.add(Conv2D(64, kernel_size=(3, 3),
          padding='same', **args))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(MaxPooling2D())
```

The code for the entire process is present in the jupyter notebook.

Refinement

Our benchmark model showed an accuracy of 0.9268

At each intermediate step, we are going to add one layer to the model, multiply the number of epochs by 4, and double the batch size.

First intermediate model: epochs = 4, batch_size = 16, 1 extra convolutional layer
Accuracy: 0.9776. About 5.48% improvement in relation to the benchmark model. Since we are not yet at 99% accuracy, we iterate again.

Second intermediate model: epocs = 16, batch_size = 32, 2 extra convolutional layers
Accuracy: 0.9960. About 1.88% improvement in relation to the previous model. We are, though, above our 99% accuracy goal. So we will save the model as the final one, and output a csv file with the predictions.

Results

Model Evaluation and Validation

The final showed more than 99,5% accuracy. A detailed summary is presented below.

Layer (type)	Output Shape	Param #
conv2d_31 (Conv2D)	(None, 32, 32, 64)	1792
batch_normalization_31 (Batc	(None, 32, 32, 64)	256
activation_31 (Activation)	(None, 32, 32, 64)	0
max_pooling2d_31 (MaxPooling	(None, 16, 16, 64)	0
conv2d_32 (Conv2D)	(None, 16, 16, 64)	36928
batch_normalization_32 (Batc	(None, 16, 16, 64)	256
activation_32 (Activation)	(None, 16, 16, 64)	0
max_pooling2d_32 (MaxPooling	(None, 8, 8, 64)	0
conv2d_33 (Conv2D)	(None, 8, 8, 64)	36928
batch_normalization_33 (Batc	(None, 8, 8, 64)	256
activation_33 (Activation)	(None, 8, 8, 64)	0
max_pooling2d_33 (MaxPooling	(None, 4, 4, 64)	0
flatten_18 (Flatten)	(None, 1024)	0
dropout_18 (Dropout)	(None, 1024)	0
dense_35 (Dense)	(None, 64)	65600
dense_36 (Dense)	(None, 1)	65

Total params: 142,081
Trainable params: 141,697
Non-trainable params: 384

As can be seen, there are three Conv2D layers that used ReLu activation. The final layers, dense_35 and dense_36, received the output of the flatten layer, progressively decreasing the dimensionality down to 1. The final model was trained using epochs = 16 and a batch size of 32. Oh the entire training set, it attained an accuracy of 0.9969. An output file with the predictions has been generated. For any entry where the probability of there being a cactus was strictly greater than 50%, the model predicted that there was a cactus in the picture. The file is predictions.csv

Justification

Comparing our final model with the initial one, we went from 92.68% to 99.69% accuracy on the training set. For our educational purposes, since we met the target of 99% or more accuracy, we can state that the final model was enough for our problem. However, it remains to be seen whether the

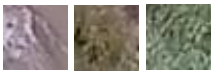
model performs well on the test set. It would need to be submitted to Kaggle for evaluation, and will be left as a future task, when the writer feels more confident about his skills in the subject.

Conclusion

Free-form visualization

For this project, our visualizations will be rather inconclusive, since the images are only 32x32 pixels. We are not going to be able to have a clear picture of our outputs. Still, the following sets of images have been chosen from the predictions file.

Images predicted by the model to contain cactus



Images predicted not to contain cactus

The inherent nature of the dataset makes it difficult for us to visualize the results properly. Zooming into the pictures greatly degrade quality, making the images even more inconclusive.

Reflection

In this project, a basic CNN was implemented to build a basic binary classifier. We started from a very limited benchmark model, and, by adding layers, increasing the batch size and the number of epochs, we managed to attain a decently high accuracy. Though the code to implement the process is relatively simple, the underlying mathematical/statistical foundations are not yet clear to me. As I tend to feel uncomfortable when I am implementing solutions whose inner workings I am not knowledgeable of, this presented a great challenge to me. On the positive side, the Keras library made it possible for me to create and tweak the models to at least attempt to solve the problem.
An interesting, yet intriguing aspect of this project, was the fact that the benchmark model was the one with the highest number of parameters: 1050735, versus 301505 of the intermediate model, and 142081 of the final model. My intuition was telling me that, as I added more layers to the model, there should be more parameters. This clearly shows my lack of foundational understanding of the subject, which will had to be dealt with by digging into theoretical literature. This was also the reason I why I opted not to submit my kernel for public evaluation on Kaggle.

Improvements

It seems likely that, if I increase the number of epochs a little more, and doubled the batch size once again, accuracy would increase. But we were already at 99.69% accuracy, so it seems reasonable that, in order to test it better, I would need to submit both the unimproved and improved models for evaluation, and verify how they fared in the actual test set. It is also possible that, using different models altogether (Resnet, SVM), I might obtain different results. Finally, I could have also performed data augmentation to see if results could be improved. But I really feel now is the time to dive a bit deeper into the foundations, and only then consider building such models.

Reference

1 - Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, Backpropagation Applied to Handwritten Zip Code Recognition; AT&T Bell Laboratories