Capstone Project

Machine Learning Engineer Nanodegree

Gabriel Verzea

October 14th, 2018

Identifying edible wild plants

Project Overview

Fuel keeps our cars running, and food is the fuel for our bodies. We can survive many days without food, but if we have to keep moving, the lack of food will slow us down and eventually bring us to a stop. I am an amateur ultrarunner and one too many times I have failed to plan, and thus found myself on a trail, away from civilization, hungry and out food. At that times, I had hoped I had shown more interest in the art of foraging, but there I was, with a lot of vegetation around me, and not knowing what was edible. I wish I had a phone app that would identify the edible plants around me...

The project aims to identify a number of 62 different wild edible plants, and once the plant is identified, provide info on the plant name and edible parts of that plant. Given a picture, it will say if the picture represents any of the 62 plants it has been trained on, or not.

The final application of this model would be to be packaged in an Android or iPhone app, that would then be able to make predictions for images taken in the wild – but this is out of scope for the current project.

A similar computer vision problem is described in the article Wäldchen, J. & Mäder, P. Arch Computat Methods Eng (2018) 25: 507. Here, a number of plants are recognized mainly based on leafs and flowers, using a number of features as Shape, Color, Texture and Leaf.

Metrics

The initial metric I had in mind, was precision. However, it seems that as of Keras 2.0, precision and recall were removed from the master branch, so I decided to stick with accuracy.

Here is how accuracy is calculated:
True positive (TP) = the number of cases correctly identified as patient
False positive (FP) = the number of cases incorrectly identified as patient
True negative (TN) = the number of cases correctly identified as healthy
False negative (FN) = the number of cases incorrectly identified as healthy

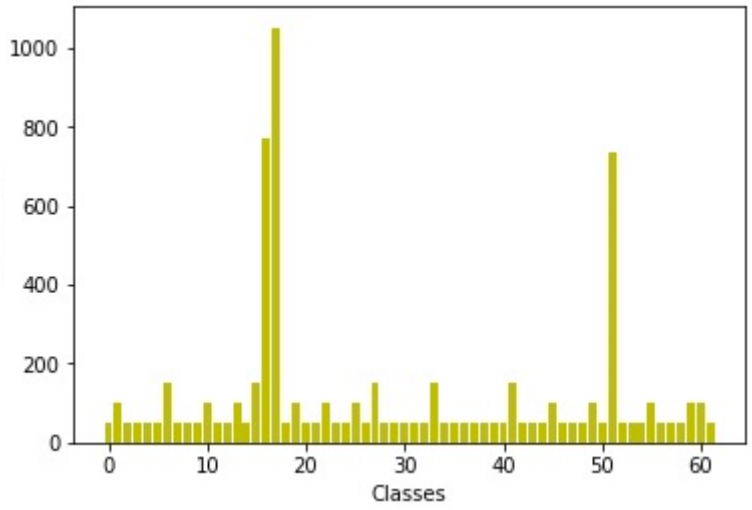
$$\text{Accuracy} = \frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}}$$

False negatives are damaging the user experience, as he misses out on plants that he could eat. But false positives could potentially be even worse, if the user is encouraged to eat a certain plant that might not really be edible. So the specificity would be more important than the sensitivity.

Dataset

As I have made up my mind on the project topic, I started looking for data. Unfortunately I could not find something readily downloadable. I did find a small flowers recognition dataset on Kaggle, with 5 plants, out of which 3 were overlapping with my classes, and I got the approval of its creator to reuse those pics. But for the rest of the 59 plant categories, I had to manually select pictures from the internet. I have decided to gather 50 images of each, and after the model was built, see if I needed to gather more, for some of the categories, where the model would be struggling. That worked fine, and I ended up with having between 50 and 150 pictures for those 59 categories.

Below is a graph with the amount of pictures per each class in the training dataset – 6558 images in total, split between 62 classes, in the dataset folder. The 3 classes where I got the free pictures from the existing Kaggle dataset clearly stand out.



For testing, I have gathered an additional 310 pictures (5 pictures per class) in the dataset-test folder. There are a few other pictures, non-labeled, that are supposed to be uploaded by the users to test the model, in the folder dataset-user_images.

Images from the edible wild plants dataset:

To keep the database size under control, I have resized all images to under 300kB.

The VGG16 model takes by default images of 224x224px, but due to the computing power restrictions of my laptop's GPU, I have decided to bring the input size even lower, to 100x100px.

The gathered pictures are in .jpg format. The color profile of a JPG image is YCbCr, which is a mathematical coordinate transformation from an associated RGB color space, according to wikipedia.

The gathered pictures will be used to train/test the model. To preserve class balances across the train/test datasets, I have used the StratifiedKFolds cross-validator. It provides train/test indices to split data in train/test sets. This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

This also allows using all the pictures in the dataset for both training and dev evaluation, by splitting the dataset into k batches, and training the model k times on k-1 batches, and evaluating it on the remaining batch, on each iteration. This is helpful when the dataset is rather limited in size, like in my case.

The dataset contains:
"dataset" folder – for training the model
"dataset-test" folder – for testing the model
"dataset-user_images" – for user uploaded images
"edible wild plants metadata.xls" – excel file with a description of the classes and the edible parts of the plants.

Preprocessing the data

The images are in .jpg format. The dimensions were not manually standardized, but this will be one step of the data preparation – to bring all the images to 100x100px. I have also made sure that no image exceeds 300KB, to keep the database within reasonable size limits.

As the dataset is gathered off the internet, the pictures come in different sizes and quality levels. At first I would need to pre-process the data. I would perform the following actions:

- Obtain a uniform aspect ratio – by cropping to squares (centered), and as well by resizing and cropping, maintaining the full image, but at a reduced scale. In this second case, padding is used, when needed.
- Perform Image scaling – scale to 100x100px
- Normalize image inputs – this makes convergence faster while training the network. Data normalization is done by subtracting the mean from each pixel, and then dividing the result by the standard deviation
- Perform Data Augmentation – generate more pictures from the existing ones, by rotation, scaling or flipping – I did this directly with Keras, when fitting the model, with the ImageDataGenerator, I have chosen to rotate images up to 20 degrees, shift them and flip them horizontally. Furthermore, for each epoch, the model was training on a number of steps_per_epoch * batch_size images, which summed up to 1200*32 = 38400 images, instead of the 6558 images in my training dataset.

I have processed the images with the help of the OpenCV library. Currently OpenCV supports a wide variety of programming languages like C++, Python, Java etc. OpenCV-Python is the Python API of OpenCV. It combines the best qualities of OpenCV C++ API and Python language.

Architecture

The solution I implemented is using deep learning, to be more specific, CNNs, to help a model learn a number of features about the training pictures. My target was that it would yield an accuracy of at least 60% on the test dataset.

A CNN network is a series convolution + pooling operations, followed by a number of fully connected layers. If we are performing multiclass classification the output is softmax

The main building block of CNN is the convolutional layer. Convolution is a mathematical operation to merge two sets of information. In our case the convolution is applied on the input data using a convolution filter to produce a feature map.

After a convolution operation we usually perform pooling to reduce the dimensionality. This enables us to reduce the number of parameters, which both shortens the training time and combats overfitting. Pooling layers down-sample each feature map independently, reducing the height and width, keeping the depth intact. I have used max pooling which just takes the max value in the pooling window. Contrary to the convolution operation, pooling has no parameters. It slides a window over its input, and simply takes the max value in the window.

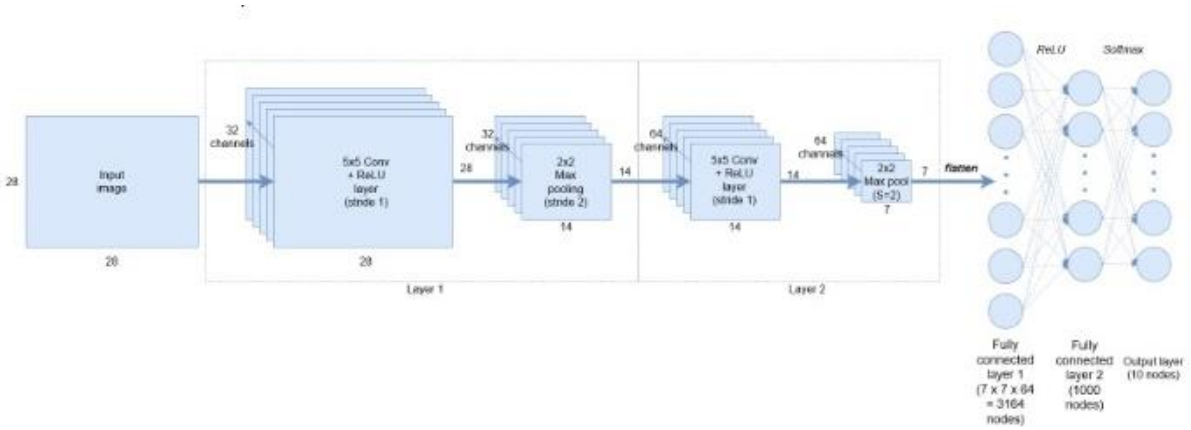
After the convolution + pooling layers we add a couple of fully connected layers to wrap up the CNN architecture.

I have built my model using transfer learning. I have chosen the VGG16 model, and I first had to import the weights of the VGG16 model, trained on ImageNet. ImageNet is a project which aims to provide a large image database for research purposes. It contains more than 14 million images which belong to more than 20,000 classes (or synsets)

I have removed the output layer, and replaced it with a series of custom layers: two Dense layers, with dropout, and a softmax layer at the end, with the 62 classes. I have added the two Dense layers with Dropout because I have noticed that the model tends to overfit quite a lot.

Benchmark model

For the benchmark model I have chosen a simple vanilla CNN.



The model contains 2 Conv layers, separated by a Pooling layer each, to reduce the number of parameters, and to make the feature detection invariant to scale and orientation changes. To connect the output of the pooling layer to the last 2 fully connected layers, we need to flatten this output into a single tensor. The last fully connected layer contains the softmax to the 62 classes.

Results

The VGG16-based model has yielded a final 99,7% accuracy on the dev dataset and a 60,6% accuracy on the test dataset.

The benchmark's model accuracy is 69,9% on the dev dataset and 14,1% on the test dataset.

The results are still indicating an overfitting on the training dataset. To address the overfitting problem, I have taken the following actions:

- I have added 2 Dropout layers at the end of the model, with 0.8 Dropout. I found this to be the optimal value for the test dataset, as I have experienced also with other values: 0.5, 0.6, 0.7 and 0.9
- I have added more data (50 to 100 more images per class), especially for the classes where the model was not predicting any of the test images correctly, or only 1.
- I have used data augmentation, to multiply the training examples, for the same purpose as the point above. Data augmentation includes things like randomly rotating the image, zooming in, adding a color filter etc. Data augmentation only happens to the training set and not on the validation/test set.
- I have tweaked he parameters of the model, I have played with different values for the learning rate (0.001 and 0.0001) and I found out that 0.0001 works the best, as

well as momentum (0.6, 0.7, 0.8, 0.9) and I found out that 0.7 works out the best

To save computing power, I have frozen the first 10 levels of the VGG-16 model. I have tried with freezing 6, 10, 14 and 18 layers, and 10 frozen layers brought the best accuracy.

When predicting the class for the user images, I am also printing the confidence level that the model had in predicting the class. If he confidence level is under 80%, I consider that the image does not represent any of the plant classes in the dataset.

I am also displaying details about the respective predicted plant, like what parts, and under which circumstances, are edible.

Ideas for further improvement

To take this application into production, the following steps can be taken, to improve test dataset accuracy:

- Increase input image sizes to the CNN model from 100x100px to 224x224px – this would take much more computing power, though
- Try to reduce overfitting by reducing the model's complexity
- Gather more training images, off the internet and doing photo-shootings in the nature

Conclusion

While the accuracy on the test set is already not too bad, this is still a prototype. Testing user images is not yet too accurate. Also, the model seems to be 100% sure of the predicted class, also when it is wrong. The model still mistakes a Lamborghini for a Daisy Fleabane. I believe there are 2 main reasons for this:

- The very limited size of the training dataset
- The very small size of the input image to the model (100x100px)

Both shortcomings can be solved if more time and resources were available.

References

Metrics
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4614595/

Image color profiles:
https://en.wikipedia.org/wiki/YCbCr
Preprocessing and reading images:
https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/

OpenCV:
https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_image_display/py_image_display.html

StratifiedKFold:
http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

Plant Species Identification Using Computer Vision Techniques:
https://link.springer.com/article/10.1007/s11831-016-9206-z

Vanilla CNN model:
http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/

Checkpoints:
https://machinelearningmastery.com/check-point-deep-learning-models-keras/

https://medium.com/@14prakash/transfer-learning-using-keras-d804b2e04ef8

Overfitting CNN models:
https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d