Improving service performance of a bird classification application

Repository https://github.com/datachord/veriff-mobile-bird-classification (shared with *khadrawy*, *rivol* and *suur* on GitHub).

This service can be experimented with Kubernetes on Katacoda. Please go to Deploy Containers Using Kubectl to sign in and start a simple Kubernetes cluster for our service.

After signing in, we can launch a single-node cluster and deploy our containerized application on this cluster.

```
minikube start --wait=false  # launch cluster
kubectl get nodes  # check cluster ready
kubectl run app --image=vinnee/mobile-bird --port 5000  # start service
kubectl get pods  # check service ready
kubectl expose deployment app --external-ip=<your service IP> --port=80
--target-port=5000  # expose service to external network
kubectl get services  # check service information
```

In the code above, <*your service IP*> will be given as you follow the steps outlined on the website. Our service is accessible at this given IP which we will later use to request service from the same machine.

To use the service, we need to provide a bird image for classification. For demonstration, we will use an image of a Phalacrocorax varius (a.k.a Australian pied cormorant). Once the image is downloaded from the link to our local machine, we can submit it to the service to identify the kind of bird shown in the image.

```
curl -o bird.jpg <image link> # download bird image
curl -X POST -F image=@bird.jpg http://<your service IP>/predict  # request
prediction
# output
# [('Phalacrocorax varius varius', 0.8430764), ('Phalacrocorax varius',
0.11654693), ('Microcarbo melanoleucos', 0.024331538)]
```

The classifier predicted the correct bird name (Phalacrocorax varius) with highest probability (0.8430764), followed by others with substantially lower probabilities.

Performance analysis

What have been done

- The class BirdClassifier was restructured to contain two methods—*load_image* and *identify*. These two methods reflect two main tasks of the classifier—acquiring image data and using image data to identify the associated kind of bird.
- Optimized reading labels (*csv* file) using *pandas* instead of parsing text, ordering predicted scores, storing predicted scores in an array instead of a nested dictionary.
- Focused the classifier on single classification rather than multiple. Multiple classifications can be achieved by calling the classifier multiple times.

Performance

New classifier takes ~5 seconds to run while the original takes ~15 seconds. Most runtime is spent on the classification task by the classification model.

Comments

- Top predictions required are few, we may not need to sort all scores; rather we can use numpy argpartition when size of list grows large. May complicate code a bit.
- If Id in label data is not important, we may use array instead of dictionary. Accessing an array may be faster than a dictionary due to the dictionary calling the hash function.
- Smallest complete operation (unit operation) of this service is the classification of a single image. Hence, the performance of the service is capped by the performance of the unit operation. In each unit operation, we can only work to improve the loading time of the image and the sorting time of the prediction outcomes. However, classification time by the model and image uploading time together may account for a larger part of the total runtime.

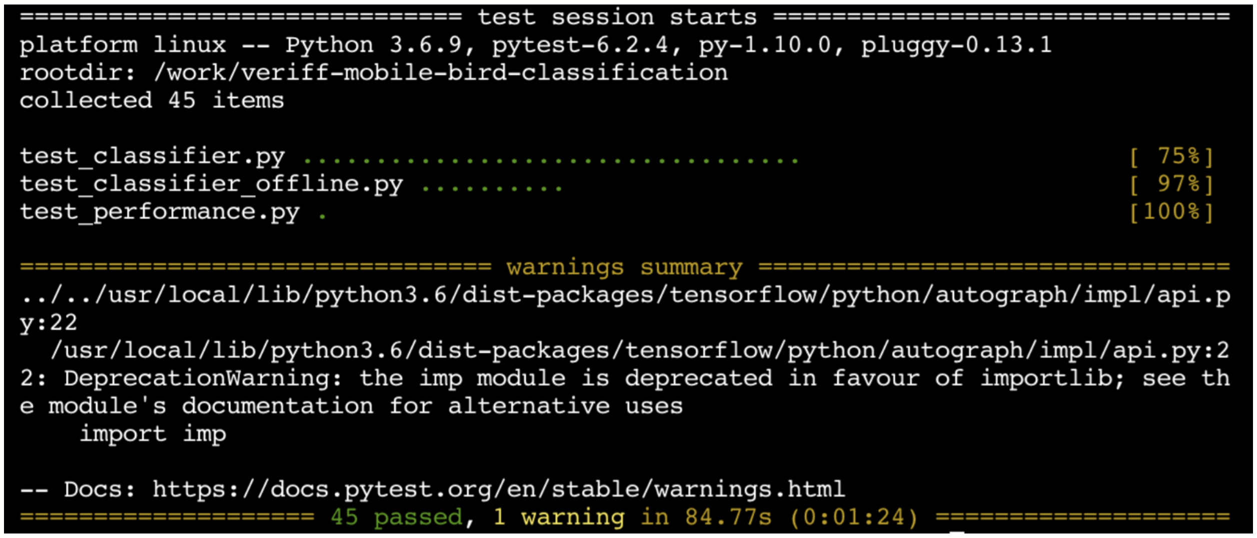
Testing logics

Testing focuses on the two methods of BirdClassifier—*load_image* and *identify*—to make sure they work properly. The *load_image* method is designed to receive image data as a file object, url or local file path, read the data into an array, then scale and normalize the data. For each of the use cases (file object/url/local file), test cases were developed accordingly with input of different types and values. The *identify* method performs classification on the image and is thus tested for accuracy.

Although the classifier is meant to fetch data online, it can also be tested offline in case of limited internet access. Such test cases rely on data on the local file system and thus can run faster than their online counterparts. Please see *prototype.ipynb* for a detailed demonstration of the code.

To run all tests, execute from the repository directory:

```
pytest
```



```
============================== test session starts ===============================
platform linux -- Python 3.6.9, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /work/veriff-mobile-bird-classification
collected 45 items

test_classifier.py .................................                       [ 75%]
test_classifier_offline.py ..........                                      [ 97%]
test_performance.py .                                                      [100%]

================================ warnings summary ================================
../../usr/local/lib/python3.6/dist-packages/tensorflow/python/autograph/impl/api.p
y:22
  /usr/local/lib/python3.6/dist-packages/tensorflow/python/autograph/impl/api.py:2
2: DeprecationWarning: the imp module is deprecated in favour of importlib; see th
e module's documentation for alternative uses
    import imp

-- Docs: https://docs.pytest.org/en/stable/warnings.html
==================== 45 passed, 1 warning in 84.77s (0:01:24) ====================
```

To check runtime:

```
pytest -s test_performance.py
# output
# Run: 1
# Top match: Phalacrocorax varius varius with score: 0.8430764079093933
# Second match: Phalacrocorax varius with score: 0.11654692888259888
# Third match: Microcarbo melanoleucos with score: 0.024331538006663322
#
# Run: 2
# Top match: Galerida cristata with score: 0.8428874611854553
# Second match: Alauda arvensis with score: 0.08378683775663376
# Third match: Eremophila alpestris with score: 0.018995530903339386
#
# ...
#
# Run: 5
# Top match: Erithacus rubecula with score: 0.8382053971290588
# Second match: Ixoreus naevius with score: 0.0030795016791671515
# Third match: Setophaga tigrina with score: 0.002611359115689993
#
# Time spent: 4.082953453063965
```

The command above calls the classifier on 5 images and prints out the top 3 predictions for each along with the total runtime of the process.

Containerize application

The Docker image for this application can be created using the Dockerfile in the repository with access to the repository. Start by creating a new folder with the Dockerfile in it and an RSA private key (*id_rsa*) outside of the new folder. The RSA private key allows us to access the remote repository with SSH. Change current directory to the new folder then run:

```
docker build --build-arg SSH_PRIVATE_KEY="$(cat ../id_rsa)" -t <image name>
. && docker rmi -f $(docker images -q --filter label=stage=app)
```

The command builds a Docker image for our application with the <*image name*> of your choice. After the build, docker will seek to remove the stage involving the private key information for security reasons.

To run a container from the image:

```
docker run -it <image name>
```

The image needs to be pushed to a registry to be accessed by Kubernetes for application deployment.

References

Deploy Your First Deep Learning Model On Kubernetes With Python, Keras, Flask, and Docker
A machine learning application that deploys to the IBM Cloud Kubernetes Service