Machine learning in Python

This guide is a concatenation of the README and a Jupyter notebook found in a git-repo here: https://github.com/epimedai/flowers. Any files referenced in the text can be found here. If you have further questions, contact me at esten@epimed.ai

Flower species classification

This repo contains the basic building blocks for creating an image classification model for recognizing flowers of various species. It exists as an extension of this machine learning workshop, hosted by Tekna. The repo consists of three main components:

1. The slides used in the workshop
2. A fully working guide implemented as a Jupyter Notebook
3. This readme, which also substitutes as a guide for setting up the necessary environment

Additionally there is a version of the guide implemented as six stand-alone python scripts, one per step, for users not familiar with Jupyter Notebooks.

Setup

The goal of this setup is creating an environment where we can run the code listed in the guide. To achieve this goal there are two necessary prerequisites:

- A working Python 3 environment with Tensorflow and Keras installed
- A folder containing the dataset structured as we want it

Both of these steps are explained here largely by virtue of a Unix terminal, but a solution for Windows users will hopefully be available soon.

Configuring the environment

Setting up our environment means installing Python and all the packages we will be needing for this project. We are in this guide going to use conda as an environment manager and pip as a package manager. There does however exist a wide variety of options out there, and as long as you are able to run the sanity check you should be good.

Installing conda

Conda is a package, dependency and environment manager for several languages, but in this project we will take advantage of the environment management capabilities. We will be using a version called Miniconda, which is installed by downloading and running a bash-script. Note that both the URL and the name of the script varies depending on your OS.

macOS: https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh Linux: https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh

```
[user@instance]:~$ wget <url>
[user@instance]:~$ sh Miniconda3-latest-<OS>-x86_64.sh
```

Running the script will trigger a bunch of prompts, one of which is

```
Do you wish the installer to prepend the Miniconda3 install location to PATH in
your /home/esten/.bashrc ? [yes|no]
```

where we recommend you answer yes. Once the installer finishes Miniconda is installed, including both Python and pip ready for use.

Creating the environment

We can create an environment with our newly installed conda installation using the command `conda create`. We do, however, have to source the .bashrc (or .bash_profile for Mac users) file modified in the previous step:

```
[user@instance]:~$ source .bashrc
[user@instance]:~$ conda create -name ml python=3.6
```

This command will create a fresh environment called "ml" with python 3.6 and a basic set of packages installed. We can activate the environment as such:

```
[user@instance]:~$ source activate ml
```

If everything went as intended the command line prompt should now be prefixed with the name of the environment.

```
[(ml) user@instance]:~$ .
```

Installing packages

The two packages necessary for this project is Tensorflow and Keras, which can both be installed by pip

```
[(ml) user@instance]:~$ pip install tensorflow
[(ml) user@instance]:~$ pip install keras
```

(Note: Users with a GPU should install tensorflow-gpu instead of tensorflow)

We also recommend installing Jupyter to be able to run the guide as a notebook:

```
[(ml) user@instance]:~$ pip install jupyter
```

Environment sanity check

We can check that everything works as it should by importing the packages in Python:

```
[(ml) user@instance]:~$ python -c “import tensorflow”
[(ml) user@instance]:~$ python -c “import keras”
```

If you are able to run these commands without anything failing horribly (warnings are OK!) you are all set up.

Note that whether or not this setup runs smoothly depends heavily on what already exists on your OS. Typical problems relate to image-specific libraries used by tensorflow. If you run into trouble you should get far by googling, or by sending me an email at esten@epimed.ai

Preparing the dataset

The dataset we will be using consists of images of flowers, 17 species with 80 samples each, and was created by the Visual Geometry Group at the University of Oxford. For later convenience we want the dataset structured as follows:

- Two folders, called train and val, each containing
- 17 folders, one per species, each containing
- A set of images of of flowers of the given species

We will be using 65 images per species for training and 15 images per species for validation.

Downloading the dataset

The dataset can be downloaded and unzipped as follows:

```
[(ml) user@instance]:~$ wget
http://www.robots.ox.ac.uk/~vgg/data/flowers/17/17flowers.tgz
[(ml) user@instance]:~$ tar -xvzf 17flowers.tgz
```

A new folder called `jpg` should appear

Restructuring

Originally the dataset is structured by having the first 80 images belonging to the first species, the next 80 images to the second, and so on. The file `restructure.py`, found in the repo, will build the structure we want if it is run in the same folder as `jpg`. If push comes to shove this step can also be done manually.

The newly created folder called `flowers` should be placed in the same directory as the guide if you want to run it. When this is the case you are ready to go!

guide

September 2, 2018

1 Introduction

In this guide we will design and train an image recognition model for recognizing 17 different species of flowers. We will use a technique called transfer learning, combining a predefined model called VGG19 trained on ImageNet with our own flower classification subnet. The guide is an extension of the workshop found here, and thus requires some shallow understanding of machine learning theory and programming in Python. Running the code has two prerequisites:

A python environment containing tensorflow and keras:

```
In [111]: import tensorflow
          import keras
```

Downloading and restructuring the dataset in a folder 'flowers' as defined here:

```
In [3]: import os

        print('Root dir exists: {}'.format(os.path.isdir('flowers')))
        print('Train dir exists: {}'.format(os.path.isdir(os.path.join('flowers', \
            'train'))))
        print('Validation dir exists: {}'.format(os.path.isdir(os.path.join('flowers', \
            'val'))))
```

```
Root dir exists: True
Train dir exists: True
Validation dir exists: True
```

2 Step 1: Serving images in-memory

To serve images to our model during training we will use a python Generator. Given that our images are structured as previously stated, a folder with the training set and a folder with the validation set, both with subfolders for each category of flowers, we can use a prebuilt keras generator called ImageDataGenerator. The generator acts as a list and serves what is known as batches of tuples, where the first element of the tuple contains the images and of the batch and the second element contains the corresponding labels.

```
In [159]: from keras.preprocessing.image import ImageDataGenerator

          generator = ImageDataGenerator()
          batches = generator.flow_from_directory('flowers/train', batch_size=4)

          batches

Found 1105 images belonging to 17 classes.


Out[159]: <keras_preprocessing.image.DirectoryIterator at 0x120ee72b0>
```

At this point it is usually a good idea to sanity check that everything works as intended. This typically includes verifying that our images are served on the correct format, and that the images and labels are still correctly matched. We can first fetch the reverse encoding of the generator to be able to decode the onehot encoded labels given in the batches

```
In [114]: indices = batches.class_indices
          labels = [None] * 17

          for key in indices:
              labels[indices[key]] = key

          labels

Out[114]: ['bluebell',
           'buttercup',
           'colts_foot',
           'cowslip',
           'crocus',
           'daffodil',
           'daisy',
           'dandelion',
           'fritillary',
           'iris',
           'lily_valley',
           'pansy',
           'snowdrop',
           'sunflower',
           'tigerlily',
           'tulip',
           'windflower']
```

We can then use matplotlib to visualize the first batch

```
In [115]: import matplotlib.pyplot as plt
          import numpy as np

          for X, y in batches:
```

```
fig, ax = plt.subplots(1, 4)

for i in range(len(X)):
    img = X[i].astype(np.uint8)
    label = labels[np.argmax(y[i])]

    ax[i].imshow(img)
    ax[i].set_title(label)
    ax[i].set_xticks([])
    ax[i].set_yticks([])

plt.show()
break # We only need the first batch
```

3 Step 2: Setting up the base model

Once we know our dataset is being served correctly we can start setting up the base model that will be the core of our flower classification model. As previously mentioned this will be a model called VGG19, a small model which yields relatively good results. Like the generator, this also exists as a prebuilt module in Keras, namely in the applications-module.

```
In [116]: from keras.applications.vgg19 import VGG19
```

When initializing the model we need to specify that we want to use the weights trained on ImageNet, that we want the entire model including top layers, and we also specify the image size we are going to use for verbosity.

```
In [118]: model = VGG19(weights='imagenet', include_top=True, input_shape=(224, 224, 3))
```

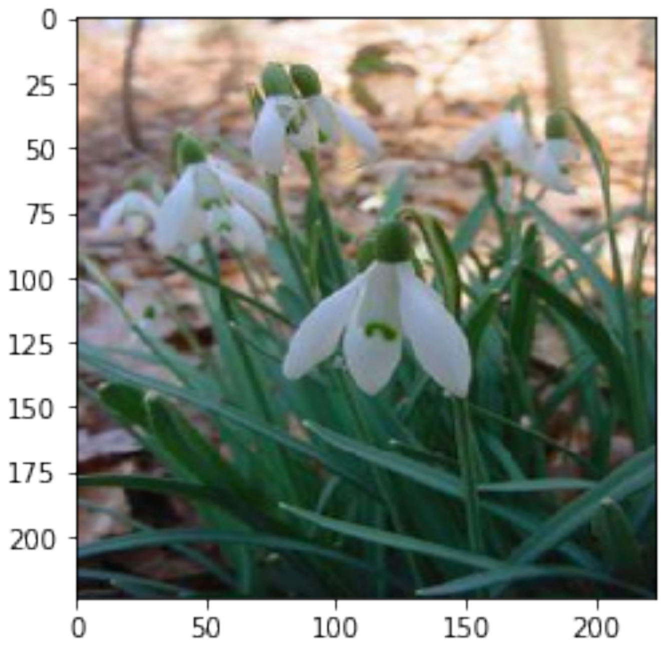
We can sanity check this step by running predicting the label for an image from our generator. Note that the predictions we are doing now will be using the labels from ImageNet, as this is what the model currently recognizes, not the labels from our dataset. We start by reinitializing the generator with the correct size and a batch size of 1. We also set the seed for the random library to control the order of the images

```
In [119]: np.random.seed(1234)
          generator = ImageDataGenerator()
          batches = generator.flow_from_directory('flowers/train', target_size=(224, 224), \
              batch_size=1)

Found 1105 images belonging to 17 classes.
```

We can then run predictions on the first batch containing a single image. To decode the prediction using imagenet labels we can use a predefined function found in the same module as the model

```
In [120]: from keras.applications.vgg19 import decode_predictions

          for X, y in batches:
              preds = model.predict(X)
              decoded_preds = decode_predictions(preds, top=1)
              fig = plt.figure()

              img = X[0].astype(np.uint8)
              label = labels[np.argmax(y[0])]
              predicted = decoded_preds[0]

              plt.imshow(img)
              fig.suptitle('Truth: {}, Predicted: {}'.format(label, predicted))
              plt.show()

              break
```

Truth: snowdrop, Predicted: [('n03991062', 'pot', 0.12648447)]



4 Step 3: Preprocessing

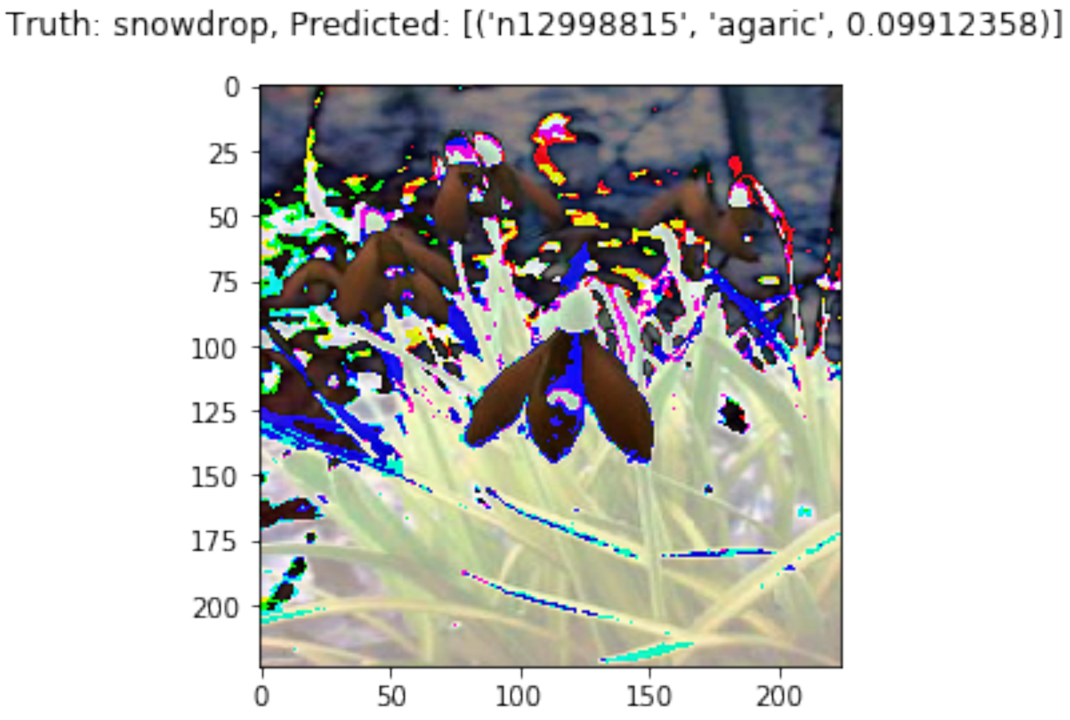
We now know both our generator and model are set up, and we are able to make predictions. The predictions, however, does not necessarily look very good. This is because of a process called preprocessing: A set of transformations applied to the images before training to give the model the best possible foundation to learn what it needs. Typical preprocessing includes rescaling the values of the data, shifting the range, and other numerical operations. Luckily, in Keras, the module which contains a model also contains the preprocessing function used for training the model. We can fetch this function and feed it to our generator to ensure all images are preprocessed before they are served to the model

```
In [121]: from keras.applications.vgg19 import preprocess_input

          np.random.seed(1234)
          generator = ImageDataGenerator(preprocessing_function=preprocess_input)
          batches = generator.flow_from_directory('flowers/train', target_size=(224, 224), \
              batch_size=1)
```

```
Found 1105 images belonging to 17 classes.
```

Once we have reinitialized the generator correctly we can rerun our predictions to see if they improve

```
In [122]: for X, y in batches:
              preds = model.predict(X)
              decoded_preds = decode_predictions(preds, top=1)
              fig = plt.figure()

              img = X[0].astype(np.uint8)
              label = labels[np.argmax(y[0])]
              predicted = decoded_preds[0]

              plt.imshow(img)
              fig.suptitle('Truth: {}, Predicted: {}'.format(label, predicted))
              plt.show()

              break
```


Truth: snowdrop, Predicted: [('n12998815', 'agaric', 0.09912358)]
0
25
50
75
100
125
150
175
200
0
50
100
150
200

Note that seeing improvements in the predictions is not a given even though the images are now preprocessed correctly. The label we are looking for might not be a part of the original dataset the model was trained on, or it might simply be a case of a bad prediction where the model misses. However, running a sanity check (preferably over more images) is usually a good habit to achieve the best results.

5 Step 4: Configuring the flower classification model

Once our we are happy with the way we are interacting with the base we can start setting up our own custom model for solving the problem we are interested in, in this case classifying flower species. The first step is to be a bit more restrictive with what we use from the pretrained model, only picking out the parts we need. We do this by dropping the top layers used for predictions, and instead perform a pooling operation on the final convolutional layer. Once we have initialized it we can fetch the input and the output of the pretrained model using properties found in keras' model class.

```
In [123]: pretrained = VGG19(include_top=False, input_shape=(224, 224, 3), weights='imagenet',
                pooling='max')
          inputs = pretrained.input
          outputs = pretrained.output
```

As we do not want the weights in this part of the final model to change, we can freeze them

```
In [124]: for layer in pretrained.layers:
                layer.trainable = False
```

We can then create our own custom layers for performing our own task. We will use a hidden fully connected layer with 128 neurons, and a final prediction layer with 17 neurons, one per specie in our dataset. Note that the hidden layer takes the output from the pretrained model as its input.

```
In [125]: from keras.layers import Dense

          hidden = Dense(128, activation='relu')(outputs)
          preds = Dense(17, activation='softmax')(hidden)
```

Once we have all our layers set up we can wrap them in a Model, and compile the model using a pretty standardized set of hyperparameters.

```
In [126]: from keras.engine import Model
          from keras.optimizers import Adam

          model = Model(inputs, preds)
          model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=1e-4), \
              metrics=['acc'])
```

We can then set up generators like we did before, one for the training data and one for validation, and train the model using fit_generator. Note that training here is set to a single epoch simply for example purposes

```
In [ ]: np.random.seed(1234)

        train_generator = ImageDataGenerator(preprocessing_function=preprocess_input)
        train_batches = train_generator.flow_from_directory('flowers/train', \
            target_size=(224, 224))

        val_generator = ImageDataGenerator(preprocessing_function=preprocess_input)
        val_batches = val_generator.flow_from_directory('flowers/val', \
            target_size=(224, 224))

        model.fit_generator(train_batches, epochs=1, validation_data=val_batches)
```

6 Step 5: Regularization

If you run the training above for a larger number of epochs, you will typically achieve a very decent result on the training data and a considerably worse outcome on the validation data. This is an example of overfitting: The model starts remembering specifics from the training set instead of learning the general features we are interested in. We handle this by introducing regularization, trying to force the model to generalize. In image recognition this is typically done using dropout-layers, which during training randomly sets the firing of a subset of neurons to 0. We can introduce dropout to our model by inserting a Dropout layer which drops 30% of the neurons between the two final layers of our model.

```
In [ ]: from keras.layers import Dropout
```

```
        hidden = Dense(128, activation='relu')(outputs)
        dropout = Dropout(.3)(hidden)
        preds = Dense(17, activation='softmax')(dropout)
```

We can recompile the model and restart training to achieve what should be a better result

```
In [ ]: model = Model(inputs, preds)
        model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=1e-4), \
            metrics=['acc'])

        np.random.seed(1234)

        train_generator = ImageDataGenerator(preprocessing_function=preprocess_input)
        train_batches = train_generator.flow_from_directory('flowers/train', \
            target_size=(224, 224))

        val_generator = ImageDataGenerator(preprocessing_function=preprocess_input)
        val_batches = val_generator.flow_from_directory('flowers/val', \
            target_size=(224, 224))

        model.fit_generator(train_batches, epochs=1, validation_data=val_batches)
```

7 Step 6: Augmentations

A second technique for avoiding overfitting is augmenting the image, which goal it is to take the existing data points in our dataset and create brand new samples. It works by somehow modifying an image in a way which changes it, while maintaining the thruthfulness of the corresponding image. An example in our case is mirroring the images vertically, which can be implemented directly in the keras generator

```
In [161]: np.random.seed(1234)

          generator = ImageDataGenerator(horizontal_flip=True)
          batches = generator.flow_from_directory('flowers/train', batch_size=1, \
              shuffle=False)
```

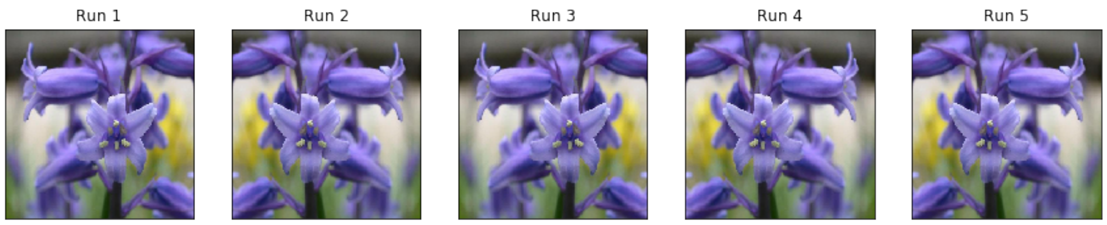
```
Found 1105 images belonging to 17 classes.
```

Using this functionality will randomly decide whether to flip the image or not each time the image is presented, theoretically yielding two samples from the single data point we started with. We can see this by visualizing the same image served from multiple batches

```
In [163]: fig, ax = plt.subplots(1, 5, figsize=(15, 10))

          for i in range(5):
              batches = generator.flow_from_directory('flowers/train', batch_size=1, \
                  shuffle=False)
```

```
        for X, y in batches:
            ax[i].imshow(X[0].astype(np.uint8))
            ax[i].set_title('Run {}'.format(i + 1))
            ax[i].set_xticks([])
            ax[i].set_yticks([])
            break

    plt.show()
```

```
Found 1105 images belonging to 17 classes.
Found 1105 images belonging to 17 classes.
Found 1105 images belonging to 17 classes.
Found 1105 images belonging to 17 classes.
Found 1105 images belonging to 17 classes.
```



Retraining the model with a set of augmentations should increase the accuracy even further.

8 Summary

The steps in this guide provide a good starting point for classifying species of flowers, or solving any other generic image classification problem. Retracing the steps while leaving more epochs for the model to train should provide a solid baseline with decent results (my best run achieved 75%). Continued work on this problem would typically include trying different architectures as core models, experimenting with various designs for the custom problem-specific final layers and testing a wide range of combinations of regularization and augmentations to combat overfitting. It should be a feasible goal to reach an accuracy in the high 90s, which seem to be how state-of-the-art models are performing. Happy hacking!