TASK

Image Processing

Visit our website

Introduction

WELCOME TO THE IMAGE PROCESSING TASK!

In this synthesis task, you'll put your programming skills to the test by developing a machine-learning model as part of your developer portfolio. Beyond simply understanding a programming language or technology, this project will help you apply that knowledge to meet real-world client specifications. Specifically, you'll build an image recognition classifier that predicts house numbers from Google Street View images. This project will not only demonstrate your ability to create practical machine-learning applications but will also showcase your development skills to potential employers.

IMAGE PROCESSING

Previously, if someone wanted to build a program to distinguish between an image of the number 1 and an image of the number 2, they would have set up a plethora of rules looking for factors like straight lines versus curly lines, or a horizontal base versus a diagonal tip, and so forth. What machine learning allows us to do instead is feed an algorithm with many examples of images that have been labelled with the correct number. The algorithm then learns for itself which features distinguish the image, and can make a prediction when faced with a new image. Typically, for a machine learning algorithm to perform well, we need lots of examples in our dataset, and the task needs to be solvable by finding predictive patterns. Let's work through an example of building an image processing model.

The dataset

We'll be using the **House Numbers dataset from Stanford University** containing images of house numbers taken from Google Street View. Each one has been cropped to 32×32 pixels in size, focused on just the number.

Cropped house number digits from Google Street View (Netzer et al., 2011)

There are a total of 531131 images in this dataset, and we will load them in as one 4D matrix of shape 32 x 32 x 3 x 531131. This represents each 32×32 image in RGB format (the red, green, and blue colour channels) for each of our 531131 images. We'll be predicting the number shown in the image, from one of ten classes (0–9). Note that in this dataset, the number 0 is represented by the label "10". The labels are stored in a 1D matrix of shape 531131 x 1. You can check the dimensions of a matrix X at any time in your program using `X.shape`.

Take note:

Although this task focuses on only house numbers, the process we will be using can be applied to any kind of classification problem. Autonomous vehicles are a huge area of application for research in computer vision at the moment, and the self-driving cars being built will need to be able to interpret their camera feeds to determine traffic light colours, road signs, lane markings, and much more. With this in mind, at the end of the task, you can think about how to expand upon what you've developed here.

The setup

To begin, we need to download the cropped digits version of the House Numbers dataset.

1. Navigate to **The Street View House Numbers (SVHN) Dataset**.
2. Under "Downloads" locate the "Format 2" section with links to the cropped digits datasets.
3. Click on the **extra_32x32.mat** (1.3 GB) dataset link and save it on your local machine in the same location as the code files for this lesson.
    a. If you have limited space on your computer you can use **train_32x32.mat** (182 MB). However, please note that this smaller dataset may lead to less optimal results.

When you've downloaded the dataset, ensure that the **image_processing_example.ipynb** Jupyter Notebook is in the same folder before continuing.

Feature processing

Now let's begin! To understand the data we're using, we can start by loading and viewing the image files. First, we need to import three libraries:

```
import scipy.io
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Then we can load the training dataset into the temporary variable `train_data`. The dictionary contains two variables X and y. X is our 4D matrix of images, and y is a 1D matrix of the corresponding labels. To access the ith image in our dataset, we would be looking for `X[:,:,:,i]`, and its label would be `y[i]`.

Let's do this for image 25:

```
# load our dataset
train_data = scipy.io.loadmat('extra_32x32.mat')

# extract the images and labels from the dictionary object
X = train_data['X']
y = train_data['y']

# reshape data to have samples in first dimension
X = X.reshape(-1, 32, 32, 3)

# view an image (e.g., 25) and print its corresponding label
img_index = 25
plt.imshow(X[img_index,...])
plt.savefig('img.png')
print(y[img_index])

# flatten to have two dimensions
X = X.reshape(-1, 32 * 32 * 3)
```

Note that if you're working in a Jupyter Notebook, you don't need to call **`plt.show()`**. Instead, call the inline function (**`%matplotlib inline`**) when you import matplotlib.

As you can see, we loaded up an image showing house number 6, and the console output from our printed label is also 6. You can change the index of the image to any number between 0 and 531130, and check out different images and their labels if you like.

However, to use these images with a machine-learning algorithm, we first need to vectorise them. This essentially involves stacking up the three dimensions of each image (the width x height x colour channels) to transform it into a 1D matrix. This gives us our feature vector, although it's worth noting that this is not really a feature vector in the usual sense. Features usually refer to some kind of quantification of a specific trait of the image, not just the raw pixels. Raw pixels can be used successfully in machine-learning algorithms, but this is typical with more complex models such as convolutional neural networks, which can learn specific features themselves within their network of layers.

Now that we have our feature vector X ready to go, we need to decide which machine-learning algorithm to use. We don't need to explicitly program an algorithm ourselves – luckily, frameworks like scikit-learn do this for us. Scikit-learn

offers a range of algorithms, with each one having different advantages and disadvantages. We won't be going into the details of each, but it's useful to think about the distinguishing elements of our image recognition task and how they relate to the choice of algorithm. This could include the amount of data we have, the type of problem we're solving, the format of our output label, etc.

For this task, we will be using a random forest approach with default hyperparameters. Since you've already learned about random forests and decision trees, you know that a random forest is essentially an ensemble of multiple decision trees. The key idea is that, rather than relying on the predictions of a single decision tree, a random forest combines the outputs of many trees and averages their results. This averaging helps reduce the risk of overfitting to the training data, making the model more robust and capable of better generalising to new data.

First, we import the necessary libraries:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Then, we define our classifier and print it to the console to see the parameter settings:

```
clf = RandomForestClassifier()
print(clf)
```

Output:

```
RandomForestClassifier(bootstrap=True, class_weight=None,
criterion='gini', max_depth=None, max_features='auto', max_leaf_nodes=None,
min_impurity_decrease=1e-07, min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1, oob_score=False,
random_state=None, verbose=0, warm_start=False)
```

Although we haven't changed any from their default settings, it's interesting to take a look at the options, and you can also experiment with tuning them. Explore the **scikit-learn documentation** to learn more about each of the parameters.

Training the model

We're now ready to train and test our model. But before we do that, we need to split our total collection of images into two sets – one for training and one for testing.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state = 42 )

clf.fit(X_train, y_train)
```

Keeping the testing set completely separate from the training set is important because we need to be sure that the model will perform well in the real world. Once trained, it will have seen many example images of house numbers. We want to ensure the model can generalise its learning to unseen number images, not just memorise the exact images it has already seen.

Because of our large dataset, and depending on your machine, this will likely take a little while to run. If you want to speed things up, you can train on less data by reducing the size of the dataset. The fewer images you use, the faster the training process, but the lower the accuracy of the resulting model.

Test results

Now we're ready to use our trained model to make predictions on new data:

```
preds = clf.predict(X_test)
print("Accuracy:", accuracy_score(y_test,preds))
```

Output:

```
Accuracy: 0.760842347049
```

Our model has learnt how to classify house numbers from Google Street View with 76% accuracy simply by showing it a few hundred thousand examples. Given a baseline measure of 10% accuracy for random guessing, we've made significant progress. There's still a lot of room for improvement here, but it's a great result from a simple untuned learning algorithm on a real-world problem. You can even try going outside and creating a 32×32 image of your own house number to test on.

Instructions

In this task, we will use the **Optical Handwritten Digits dataset** that was generated using NIST's preprocessing program (Alpaydin and Kaynak, 1998). It consists of normalised bitmaps of handwritten digits. These digits were collected from 43 individuals, with 30 contributing to the training set and 13 to the testing set. The original 32x32 bitmaps were downscaled to 8x8 matrices.

Read and run the Jupyter Notebook in this task's folder before attempting the practical task.

Practical task

Create a notebook called **optdigits_task.ipynb** and follow the steps below:

- Load the **optdigits** data into your Jupyter Notebook.
- Split the data into train and test sets.
    - Add a comment explaining the purpose of the train and test sets.
- Use the scikit-learn `RandomForestClassifier` to create a classification model.
- Pick one parameter to tune, and explain why you chose this parameter.
- Select a value for the parameter to use during testing on the test data, and provide a rationale for your choice.
- Output a confusion matrix for your model on the test set.
- Based on the confusion matrix, report the class with the highest number of misclassifications, if any.
- Report the accuracy, precision, recall, and f1-score. Hint: use `average="macro"` in `precision_score`, `recall_score`, and `f1_score` from scikit-learn.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

REFERENCE LIST

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., & Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. **http://ufldl.stanford.edu/housenumbers/**