Hyperiondev

TASK

Image Processing

Visit our website

Introduction

WELCOME TO THE IMAGE PROCESSING SYNTHESIS TASK!

SYNTHESIS TASKS

Synthesis tasks allow you to test your programming skills while creating a developer portfolio. Synthesis tasks are projects or challenges similar to your other tasks but they are entirely for learning – they are not graded. They enable you to gain experience and to showcase that experience through an addition to your developer portfolio. It is essential for developers to understand a programming language or technology. However, it is even more important to be able to apply your knowledge to create software that meets unique client specifications. Creating software allows you to highlight your development skills to a prospective employer!

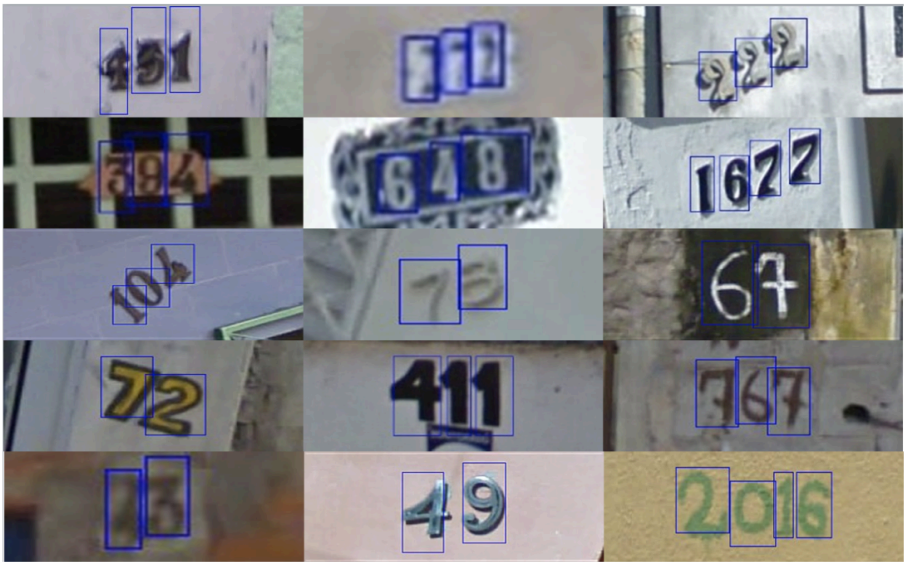
For this synthesis task, we'll be building an image recognition classifier that accurately determines the house number displayed in images from Google Street View.

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 vs curly lines, or a horizontal base vs 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 one which is solvable through finding predictive patterns.

The dataset

For this Practical Task example, we'll be using the House numbers dataset from Stanford University (**http://ufldl.stanford.edu/housenumbers**). It contains 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.



Source: http://ufldl.stanford.edu/housenumbers

There are a total of 531,131 images in this dataset, and we will load them in as one 4D-matrix of shape 32 x 32 x 3 x 531,131. This represents each 32×32 image in RGB format (the three red, green, and blue colour channels) for each of our 531,131 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 531,131 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 just 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 **extra_32x32.mat** dataset and save it in our working directory. This is a large dataset (1.3 GB in size). If you don't have sufficient space on your computer, you may alternatively use the **train_32x32.mat** dataset which is smaller (182MB). However, please be aware that using the reduced dataset may lead to less optimal results due to the smaller amount of data.

Note: If the provided links do not open properly, please copy the URL portion from the redirection notice and paste it directly into your browser. Once you hit enter/go, the download will automatically commence.

When you've downloaded the dataset, create a new Jupyter notebook and name it **image_processing.ipynb**.

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
```

Then we can load the training dataset into a 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 i-th 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, use the inline function (*`%matplotlib inline`*) just once when you import matplotlib.*

As you can see, we load up an image showing house number 3, and the console output from our printed label is also 3. You can change the index of the image (to any number between 0 and 531,130) 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.

We will be using a Random Forest approach with default hyperparameters. You can learn more about Random Forests **here**, but in brief they are a construction of multiple decision trees with an output that averages the results of individual trees to prevent fitting too closely to any one tree.

First, we import the necessary library and then define our classifier:

```
from sklearn.ensemble import RandomForestClassifier

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

We can also print the classifier to the console to see the parameter settings used. Although we haven't changed any from their default settings, it's interesting to take a look at the options and you can experiment with tuning them.

```
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)
```

Training the model

We're now ready to train and test our data. But before we do that, we need to split our total collection of images into two sets – one for training and one for testing. 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 be sure that when presented with new images of numbers it hasn't seen before (the testing set), it has actually learnt something from the training and can generalise that knowledge – not just remember the exact images it has already seen.

```
from sklearn.model_selection import train_test_split

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)
```

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:

```
from sklearn.metrics import accuracy_score

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 **MNIST database** (LeCun et al., 2010). Its creators describe it as follows: *"The MNIST database of handwritten digits ... has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalised and centred in a fixed-size image."*

First read and run the **MNIST.pynb** example file to explore the MNIST hand-written dataset provided by scikit-learn. Then, follow the instructions below to create a Random Forest model using the sample of the MNIST data provided by scikit-learn.

Practical Task 1

Follow these steps:

- Create a copy of the **MNIST.ipynb** file and rename it **mnist_task.ipynb**.
- Load the MNIST dataset. Use a library such as scikit-learn to access the dataset (`from sklearn.datasets import load_digits`).
- Split the data into train and test sets.
  - Add a comment explaining the purpose of the train and test sets.
- Use the RandomForestClassifier built into scikit-learn 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.

- Print the confusion matrix for your Random Forest model on the test set.
- Report which classes the model struggles with the most.
- Report the accuracy, precision, recall, and f1-score. *Hint: use average="macro" in precision_score, recall_score and f1_score from scikit-learn.*

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCES

LeCun, Y., Cortes, C. and Burges, C.J.C. (2010). The MNIST Database of Handwritten Digits, Available from **http://yann.lecun.com/exdb/mnist/**