Practical 3: Using Pre-trained ConvNets
3rd IAPR Summer School on Document Analysis
August 2019

Introduction

Transfer learning refers to the technique of using knowledge of one domain to another domain i.e. a Neural Network model trained on one data-set can be used for other data-set by fine-tuning the former network. This is a function of several factors, but the two most important ones are the size of the new dataset, and its similarity to the original dataset (e.g. ImageNet-like in terms of the content of images and the classes, or very different).

Task 1: Using a Pre-Trained ConvNet for Image Classification

In the first task, you will classify an object using the pre-trained VGG model trained on 1000 classes of the ImageNet dataset. If the new dataset has the same classes as the training dataset, then the pre-trained CNN can be used directly to predict the class of the images from the new dataset.

VGG released two different CNN models, specifically a 16-layer model and a 19-layer model. The VGG model can be loaded and used in the Keras deep learning library. Keras provides an Applications interface for loading and using pre-trained models. Using this interface, you can create a VGG model using the pre-trained weights provided by the Oxford group and use it as a starting point in your own model, or use it as a model directly for classifying images.

We can use the standard Keras tools for inspecting the model structure. For example, you can print a summary of the network layers as follows.

```
from keras.applications.vgg16 import VGG16
model = VGG16()
print(model.summary())
```

You can load an image and resize it to the desired network size. The network expects one or more images as input; that means the input array will need to be 4-dimensional: samples, rows,

columns, and channels. We only have one sample (one image). We can reshape the array by calling reshape() and adding the extra dimension. Next, the image pixels need to be prepared in the same way as the ImageNet training data was prepared. Keras provides a function called preprocess_input() to prepare new input for the network.

Figure 1 Coffee Mug Image fed to VGG 16 for classification

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16
# load the model
model = VGG16()

# load an image from file
image = load_img('mug.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
```

We can call the predict() function on the model in order to get a prediction of the probability of the image belonging to each of the 1000 known object types.

```
# predict the probability across all output classes
yhat = model.predict(image)
```

It can return a list of classes and their probabilities in case you would like to present the top 3 objects that may be in the photo. We will just report the first most likely object.

```
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
# One list of probable objects for each query in batch is returned
# Since we have only one query, only one list is returned
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

Running the example, we can see that the image is correctly classified as a “coffee mug” with a 75% likelihood.

Task 2: Using a Pre-Trained ConvNet as Feature Extractor

As discussed earlier, Keras comes bundled with many models. A trained model has two parts – Model Architecture and Model Weights. The weights are large files and thus they are not bundled with Keras. However, the weights file is automatically downloaded (one-time) if you specify that you want to load the weights trained on ImageNet data. It has the following models (as of Keras version 2.1.2):

- VGG
- InceptionV3
- ResNet
- MobileNet
- Xception
- InceptionResNetV2

You can load any of the models in Keras using the following.

```
import numpy as np
from keras.applications import vgg16, inception_v3, resnet50, mobilenet

#Load the VGG model
vgg_model = vgg16.VGG16(weights='imagenet')

#Load the Inception_V3 model
inception_model = inception_v3.InceptionV3(weights='imagenet')

#Load the ResNet50 model
resnet_model = resnet50.ResNet50(weights='imagenet')

#Load the MobileNet model
mobilenet_model = mobilenet.MobileNet(weights='imagenet')
```

For this example, you will be working on the VGG model.

Step 1: Load the model:

```
#Include_top=False, Does not load the last two fully connected layers which act as the classifier.
#We are just loading the convolutional layers.
#It should be noted that the last layer has a shape of 7 x 7 x 512.
vgg_conv = VGG16(weights='imagenet',include_top=False,input_shape=(224,224,3))
```

Step 2: Load Data and Labels:

Data is assumed to be a 4D tensor: (Samples, 224, 224, 3)

```
X_train,X_test,Y_train,Y_test = train_test_split(data,labels, test_size=0.20, random_state=42)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

Step 3: Convert Labels to One-hot-Encoding:

```
# convert class vectors to binary class matrices
Y_train = keras.utils.to_categorical(Y_train, num_classes)
Y_test = keras.utils.to_categorical(Y_test, num_classes)
```

Step 4: Create Tensors to store Features:

```
nTrain = X_train.shape[0]
nTest = X_test.shape[0]

train_features = np.zeros(shape=(nTrain,7,7,512))
test_features = np.zeros(shape=(nTest,7,7,512))
```

Step 5: Pass images through network using predict function to get features:

```
train_features = vgg_conv.predict(X_train)
train_features = np.reshape(train_features, (nTrain, 7 * 7 * 512))

test_features = vgg_conv.predict(X_test)
test_features = np.reshape(test_features, (nTest, 7 * 7 * 512))
```

It is important to note that dropping the fully connected layers and keeping the convolutional base returns a volume of 7x7x512 which can be flattened to a feature vector of size 25,088. It is also possible to extract features from the fully connected layer (dimension: 4096). Keras Functional API can be used for this purpose.

Step 6: Classifier Training and Evaluation:

Use any classifier employing training and test features.

Task 3: Fine Tuning a Pre-Trained ConvNet

The task of fine-tuning a network is to tweak the parameters of an already trained network so that it adapts to the new task at hand. As explained previously, the initial layers learn very general features and as we go higher up the network, the layers tend to learn patterns more specific to the task it is being trained on. Thus, for fine-tuning, we want to keep the initial layers intact (or freeze them) and retrain the later layers for our task.

Step 1: Load the Model

First, we will load a VGG model without the top layer (which consists of fully connected layers).

```
#Include_top=False, Does not load the last two fully connected layers which act as the classifier.
#We are just loading the convolutional layers.
vgg_conv = VGG16(weights='imagenet',include_top=False,input_shape=(224,224,3))
```

It is also possible to change the size of input when loading only convolutional base of the network. For instance:

```
vgg_conv = VGG16(weights='imagenet',include_top=False,input_shape=(100,100,3))
```

Step 2: Freeze the Initial Layers:

In Keras, each layer has a parameter called "trainable". For freezing the weights of a particular layer, we should set this parameter to False, indicating that this layer should not be trained.

```
for layer in vgg_conv.layers[:-4]:
    layer.trainable=False
```

Step 3: Create a New Model

Now that we have set the trainable parameters of our base network, we would like to add a classifier on top of the convolutional base. We will simply add a fully connected layer followed by a softmax layer with as many outputs as the number of layers.

```
model = Sequential()
# Add the vgg convolutional base model
model.add(vgg_conv)
# Add new layers
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
 # Show a summary of the model. Check the number of trainable parameters
model.summary()
```

Step 4: Train and Evaluate the model

The network can now be trained using any dataset. The last four convolutional and the newly added fully connected layers are trained while the initial layers employ the weights of the ImageNet dataset.

++++++++++