Classifying SceneSet15

Nikhil Santokhi

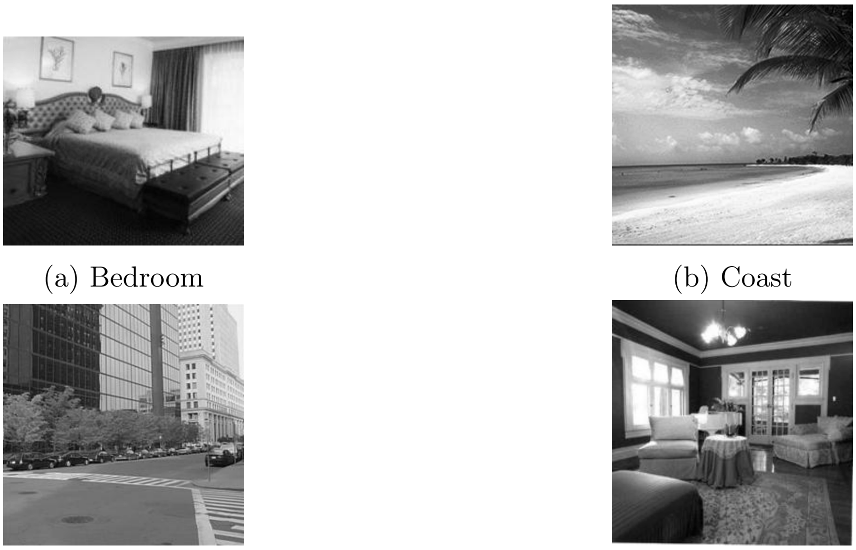
September 2024

Contents

1 SceneSet15 Dataset

The SceneSet15 dataset consists of 15 different classes based on a variety of environments and settings covering both indoor and outdoor spaces, as well as natural and urban scenes.

This dataset is small, having only *100 images per class* for a total of 1500 images.



(a) Bedroom

(b) Coast

(c) Street

(d) Living Room

Figure 1: Examples of images from the Bedroom, Coast, Street and Living Room class.

The target accuracy for classifying this dataset is *80%* or more. We can achieve this by fine-tuning a pre-trained network, which already has initialised weights from being pre-trained on ImageNet.

Fine-tuning is applied to a pre-trained state-of-the-art architectures such as VGG16, ResNet50, Inception and Xception that have all been trained on the ImageNet dataset. To perform fine-tuning you create a *custom built fully-connected head* and replace the network's own head, while keeping the original body.

But since the body's `CONV` layers have already been trained they have learnt the discriminative filters whereas the *new head hasn't.* We combat this problem by allowing the head to have a 'warm up' phase where the rest of the architecture is 'frozen' to allow the randomly initialised `FC` head to learn the discriminative patterns we seek. Also for small datasets (which it is in our case) it can be difficult for a network to start learning from a 'cold' start which is why we have to freeze the body.

Once these patterns have been learnt, we unfreeze the body then continue training until a sufficient accuracy is acquired. 'What does freezing a network actually mean?', it is when training data is propagated forwards through the network (as normal) but only backpropagated through the `FC` layers then stops after that.

Why not use a custom architecture? As mentioned before fine-tuning only works on pre-trained architectures, and using it will result in much better results than a custom made network. This is because fine-tuning is a super powerful method, which can most of the time guarantee much higher accuracy. This is why we will be using VGG16 as it is pre-trained on ImageNet and its ability to extract high-level features from images allowing for precise classification.

2 VGG16 on SceneSet15

Firstly set up the project folder as so:

```
Project
├── dataset
│   └── sceneset15
│       └── ...
├── utilities
│   ├── __init__.py
│   ├── aspectawarepreprocessor.py
│   ├── fcheadnet.py
│   ├── imagetoarraypreprocessor.py
│   └── simpledatasetloader.py
└── finetune_sceneset15.py
```

Now open `finetune_sceneset15.py`, and insert the following code:

```
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import classification_report
5  from utilities import ImageToArrayPreprocessor
6  from utilities import AspectAwarePreprocessor
7  from utilities import SimpleDatasetLoader
8  from utilities import FCHeadNet
9  from tensorflow.keras.preprocessing.image import ImageDataGenerator
10 from tensorflow.keras.optimizers import RMSprop
11 from tensorflow.keras.optimizers import SGD
12 from tensorflow.keras.applications import VGG16
13 from tensorflow.keras.layers import Input
14 from tensorflow.keras.models import Model
15 from imutils import paths
16 import numpy as np
17 import argparse
18 import os
```

Lines 2-18 import all the packages we will need. **Lines 5-7** import our image preprocessors to make them viable for VGG16 along with a dataset loader for SceneSet15. **Line 8** is the new network head in which we will being replacing VGG16's current head and **Line 9** shows that we are going to use data augmentation.

Lines 10 and 11 are the two optimisers we will be using, RMSprop in the first section because the warm up stage can pose a challenge and require different optimisers than SGD, which we will be using for the rest of the training.

Lines 13-18 imports all other useful packages/classes we will need to use.

```
20 # construct the argument parse and parse the arguments
21 ap = argparse.ArgumentParser()
22 ap.add_argument("-d", "--dataset", required=True,
23                 help="path to input dataset")
24 ap.add_argument("-m", "--model", required=True,
25                 help="path to output model")
26 args = vars(ap.parse_args())
```

Here we are parsing two arguments:

- `--dataset`: the path to the input SceneSet15 dataset.
- `--model`: the path to our output model of serialised weights after training.

```
28 # grab the list of images that we'll be describing, then extract
29 # the class label names from the image paths
30 print("[INFO] loading images...")
31 imagePaths = list(paths.list_images(args["dataset"]))
32 classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]
33 classNames = [str(x) for x in np.unique(classNames)]
```

This code block is responsible for assembling the images and extracting their respective class labels from disk.

```
35 # initialise the image preprocessors
36 aap = AspectAwarePreprocessor(224, 224)
37 iap = ImageToArrayPreprocessor()
38
39 # load the dataset from disk then scale the raw pixel intensities to
40 # the range [0, 1]
41 sdl = SimpleDatasetLoader(preprocessors=[aap, iap])
42 (data, labels) = sdl.load(imagePaths, verbose=500)
43 data = data.astype("float") / 255.0
```

Lines 36 and 37 initialise the image preprocessors. `AspectAwarePreprocessor` is used to resize each image to 224 x 224 pixels (which is a prerequisite for VGG16), while retaining each images its aspect ratio so they don't become distorted in any way.

Lines 41-43 loads the dataset images and labels, applies the image preprocessors and then finally scales the raw pixel intensities into the range [0, 1] this is done to preserve the relative order and distance between data points all while making it easier to utilise.

```
45 # partition the data into training and testing splits using 75% of
46 # the data for training and the remaining 25% for testing
47 (trainX, testX, trainY, testY) = train_test_split(data, labels,
48                 test_size=0.25, random_state=42)
49
50 # convert the labels from integers to vectors
51 trainY = LabelBinarizer().fit_transform(trainY)
52 testY = LabelBinarizer().fit_transform(testY)
```

This part is simple, **Lines 47 and 48** are simply creating our training and testing split of 75% training and 25% testing. Then one-hot encoding the labels on **Lines 51 and 52** which means turning the labels from integer labels to vector labels so it's easier to learn.

```
54 # construct the image generator for data augmentation
55 aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
56             height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
57             horizontal_flip=True, fill_mode="nearest")
```

Here we are adding data augmentation which applies geometric transformations to aid the training data by increasing it, here it is extremely useful as we are dealing with a not so large dataset.

```
59 # load the VGG16 network, ensuring the head FC layer sets are left
60 # off
61 baseModel = VGG16(weights="imagenet", include_top=False,
62                   input_tensor=Input(shape=(224, 224, 3)))
63
64 # initialise the new head of the network, a set of FC layers
65 # followed by a softmax classifier
66 headModel = FCHeadNet.build(baseModel, len(classNames), 256)
67
68 # place the head FC model on top of the base model -- this will
69 # become the actual model we will train
70 model = Model(inputs=baseModel.input, outputs=headModel)
```

Lines 61 and 62 initialise the VGG16 network using the pre-trained weights from ImageNet and leaving off the head of the network (`include_top=False`) so we can add our own. Then defining the correct image dimensions of 224 x 224 x 3 pixels which is desired by VGG16 as previously stated.

Line 66 creates the new head of the network using `FCHeadNet` which takes the `baseModel` body as an input, `len(classNames)` as the number of class labels which in our case is 15, and then `256` nodes in the FC layer of the head.

Line 70 is where our new version of VGG16 is assembled, taking the `baseModel` body as the input and `headModel` as the output.

```
72 # loop over all layers in the base model and freeze them so they
73 # will *not* be updated during the training process
74 for layer in baseModel.layers:
75     layer.trainable = False
```

As we are fine-tuning the network we need to freeze *all* of the body to allow the head of the network to warm up in the first stage of training.

```
77 # compile our model (this needs to be done after setting our
78 # body layers to being non-trainable)
79 print("[INFO] compiling model...")
80 opt = RMSprop(learning_rate=0.001)
81 model.compile(loss="categorical_crossentropy", optimizer=opt,
82               metrics=["accuracy"])
83
84 # manually creating a validation split from the training dataset
85 (trainX, valX, trainY, valY) = train_test_split(trainX, trainY,
86               test_size=0.15, random_state=42)
87
88 # train the head of the network for a few epochs (all other
89 # layers are frozen) -- this will allow the new FC layers to
90 # start to become initialised with actual "learned" values
91 # versus pure random
92 print("[INFO] training head...")
```

```
93  model.fit(aug.flow(trainX, trainY, batch_size=64),
94             validation_data=(valX, valY), epochs=25,
95             steps_per_epoch=len(trainX) // 64, verbose=1)
```

Line 80 sets the optimiser to be RMSprop with a learning rate of 0.001, the reason for such a low learning rate is because the head of the network needs to warm up and you don't want it to have a larger than needed affect on the final result.

Lines 85 and 86 split the already made 75% training set into 85% training and 15% validation. So the model can be tested against unseen data.

Lines 92-95 train our new head using the data augmentation variable we made earlier, we are training the head for 25 epochs before unfreezing the body.

```
97  # evaluate the network after initialisation
98  print("[INFO] evaluating after initialisation...")
99  predictions = model.predict(testX, batch_size=64)
100 print(classification_report(testY.argmax(axis=1),
101            predictions.argmax(axis=1), target_names=classNames))
```

This is to show the accuracy after training the head so we can later compare it to the results after fine-tuning and see how much it improves by.

```
103 # now that the head FC layers have been trained/initialised, lets
104 # unfreeze the final set of CONV layers and make them trainable
105 for layer in baseModel.layers[15:]:
106     layer.trainable = True
```

As VGG16 is a deep network we will only be unfreezing the top `CONV` layer then continue training.

```
108 # for the changes to the model to take effect we need to recompile
109 # the model, this time using SGD with a *very* small learning rate
110 print("[INFO] re-compiling model...")
111 opt = SGD(learning_rate=0.001)
112 model.compile(loss="categorical_crossentropy", optimizer=opt,
113               metrics=["accuracy"])
114 
115 # train the model again, this time fine-tuning *both* the final set
116 # of CONV layers along with our set of FC layers
117 print("[INFO] fine-tuning model...")
118 model.fit(aug.flow(trainX, trainY, batch_size=64),
119            validation_data=(valX, valY), epochs=100,
120            steps_per_epoch=len(trainX) // 64, verbose=1)
```

Now we use SGD (still with a very small learning rate for the same reason as before) and train for 100 epochs to allow the unfrozen `CONV` layer to adjust to the underlying patterns in the training data.

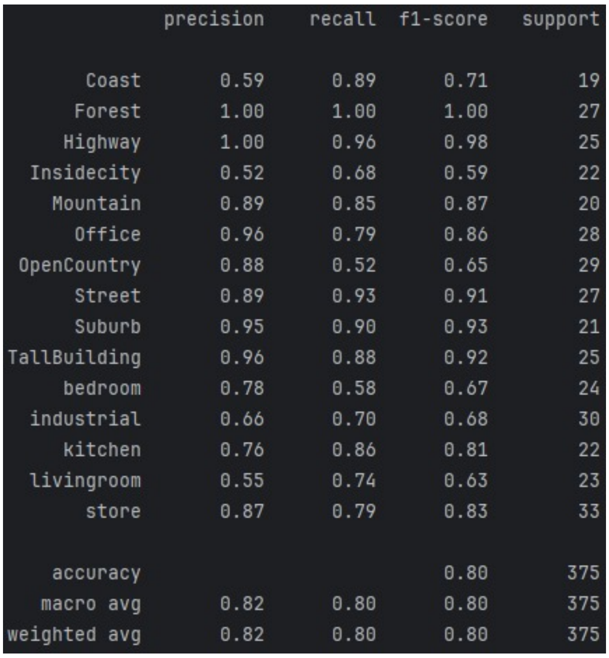
```
122 # evaluate the network on the fine-tuned model
123 print("[INFO] evaluating after fine-tuning...")
124 predictions = model.predict(testX, batch_size=64)
125 print(classification_report(testY.argmax(axis=1),
                predictions.argmax(axis=1), target_names=classNames))

# save the model to disk
print("[INFO] serialising model...")
model.save(args["model"])
```

Lastly we evaluate the fine-tuned network after training is completed and serialise the weights to disk, so we can test the model at a later time.

To run the program execute the following command: `python finetune_sceneset15.py --dataset ./dataset/sceneset15 --model output/sceneset15.keras`

3 Results



```
              precision    recall  f1-score   support

       Coast       0.59      0.89      0.71        19
      Forest       1.00      1.00      1.00        27
     Highway       1.00      0.96      0.98        25
  Insidecity       0.52      0.68      0.59        22
    Mountain       0.89      0.85      0.87        20
      Office       0.96      0.79      0.86        28
 OpenCountry       0.88      0.52      0.65        29
      Street       0.89      0.93      0.91        27
      Suburb       0.95      0.90      0.93        21
TallBuilding       0.96      0.88      0.92        25
     bedroom       0.78      0.58      0.67        24
  industrial       0.66      0.70      0.68        30
     kitchen       0.76      0.86      0.81        22
  livingroom       0.55      0.74      0.63        23
       store       0.87      0.79      0.83        33

    accuracy                           0.80       375
   macro avg       0.82      0.80      0.80       375
weighted avg       0.82      0.80      0.80       375
```

Figure 2: Results on SceneSet15 after fine-tuning VGG16.

As you can see my results have only just achieved the target of *80%*.

More coming soon...

4 Summary

Since the accuracy target was only just achieved it shows that maybe a deeper network such as ResNet50 would achieve a higher accuracy for this data set.

ResNet50 did not produce good results further investigation needed as to why.