Keras Notes from here

5.3-using-a-pretrained-convnet

在处理小数据集的问题上一个最常用和有效的方法是利用现有的已经训练好的网络模型。**这个预训练模型一般是在一个大的数据集上进行大规模分类任务保存下来的。如果这个数据集非常大并且内容非常普遍，那么这个预训练的模型从中得到的特征映射层就可以最为我们可视化世界的通用模型，所以它的特征也可以有效的用于不同的计算机视觉分类问题，哪怕这个分类认为与原始分类认为完全不同。**例如我们可以运用别人在ImageNet（通常都是些动物和日常用品）上训练好的网络，重新设计一下用于一个与原来毫无关系的家具分类任务。与许多过去的浅层的方法相比，深度学习的一个关键优势在于在不同问题中学习到的特征的可移植性，这就使得深度学习在涉及小数据集的问题上非常有效。

在我们的这个例子中，我们将考虑一个在Imagenet数据集上训练的大型的卷及网络（140万个标记的图像数据和1000个不同的类别）。ImageNet包含许多不同的动物类别，其中就是多种不种类的猫和狗，所以我们期待它在我们的猫狗分类上有不俗的表现。我们将使用由Karen Simonyan和 Andrew Zisserman在2014年开发的VGG16体系结构，这是一种简单而广泛使用的用于ImageNet分类的结构。虽然这个模型有点老，并且和当今最先进的模型相比有点差距，并且与其他模型相比有点笨重，我们选中它的原因是这个模型与我们之前使用的模型有相似之处，并且不用介绍新的概念，容易理解。这或许是许多网络结构模型中的一个，其他的还有-VGG，ResNet, Inception, Inception-RresNet, Xception…,你要慢慢习惯这些属于，如果你利用深度学习进行计算机视觉方面的工作，你会经常用到他们。

使用一个预训练的模型包括两部：特征提取和fine-tuning。我们这里都有，让我们先开始特征提取

特征提取

特征提取包括使用先前网络学习到的表示方法从新的样本中提取有意义个特征。之后这些特征在经过一个从头开始训练得到的分类器，进行分类。正如我们先前看到的那样，用于图像分类的卷积网络有两部分组成：前部分是由一系列的卷积池化层组成，后一部分是一个全连接层的分类器。**特征提取层只是简单的将新的数据通过之前的网络结构，并在网络最后的输出上训练一个新的分类器。**

为什么只是用卷积结构层？之前网络的分类还能使用吗？一般而言，我们不会这么做。原因很简单，卷积网络结构学习到的特征表示更具有通用性，所以更加可用。卷积网络结构的特征映射是一幅图像更加通用的呈现，因此对于计算机视觉问题更加适用。与之相对应的是分类器学习到的特征表示非常特定于训练集的类别-它只包含这个图像在这个类中的概率信息。除此之外，全连接层得到的特征表示不在包含输入图像的位置信息，也就是说丢失了空间信息，但是卷积层特征却仍然存在这些信息。因此对于需要对象位置的问题，全连接层一般是没用的。**模型的前几层一般提取局部的，通用性比较高的特征映射（比如视觉边缘，颜色，纹理等），然而后面几层提取的是更抽象的特征概念（比如猫的耳朵，狗的眼睛等）。所以如果新的数据集与模型原来训练的数据集差异很大的话，你最好使用模型的前面几层作为特征提取器，而不是使用整个卷积网络。**

在我们这个例子中，因为ImageNet分类数据集中包含了各种类型的猫和狗，所以重新使用包含全连接层的原始网络结构模型是非常好的。但是为了使例子更加具有普遍性，也就是分类任务的类

别集与ImageNet的分类任务类别集不存在重叠部分的问题，我们不会选择使用整个原始模型结构。

下面让我们用VGG16网络结构在猫狗数据集上提取有意义的特征，并且在这些特征的基础上训练一个猫狗分类器。

VGG16网路结构，包括上面提到的其他网络结构模型，都可以通过keras.applications进行导入。下面的代码用来实例化一个VGG16网络结构模型。

```
from keras.applidations import VGG16
conv_basee = VGG116(weights='imagenet',
                    include_top=False,
                    input_shape=(150, 150, 3))
```

我们传递了三个参数：

- weights：用来初始化模型
- include_top：用于指定是否包含全连接层，默认情况下，这个全连接层是对应于ImageNet的1000分类器，因为我们要自己训练两类的分类器，所以这个top分类器并不需要。
- input_shape:用于输入网络的Tensor的shape，这个参数是可选的，如果不设置的话，那么网络可以接受任意shape的输入。
  你可以使用下面的代码查看网络整体结构。
  `conv_base.summary()`

这个网络由两种conv-pooling组成，一种是两个卷积层加一个最大pooling层，共有两个，另一种是三个卷积层加一个最大pooling层，共有三个，最后的输出是（4,4,512），我们要在这个特征输出上堆叠一个全连接层分类器。
现在我们有两条路可以走：

- 在数据集上运行VGG16的卷积结构，将输出结果保存成Numpy array的形式，之后将这些数据输入densely-connected 分类器。这个过程非常简单快捷，因为输入的每一幅图像都只进行一次卷积操作。这样操作的话有一个弊端就是不能对数据进行augmentation（？）。
- 在VGG16卷积结构上扩展一个全连接层的，以端到端的形式运行整个输入数据集。这样可以进行augmentation，因为每次输入的图像都是通过卷积来进行的（？）。这种方法比第一种方法更昂贵（？）。
  这两种方法我们都会尝试，首先是第一种方法，记录卷积网络结构的输出，并将这些输出作为新模型的输入。
  我们使用之前提到的ImageDataGenerator提取图像和标签，并保存成numpy array的形式。之后调用VGG16模型的predict方法提取整个数据集上的特征。

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'

train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1./255)
```

```
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            # Note that since generators yield data indefinitely in a loo
p,
            # we must `break` after every image has been seen once.
            break
    return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir,
 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

现在我么提取到的特征的形状是（samples, 4, 4, 512），我们要将这个数据输入densely-connected classifier,所以我们需要先将他们拉成（samples， 8192）。

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 *
512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))
```

现在我们需要定义一个densely-connected classifier(注意其中使用了dropout用于对模型进行正则化)，并利用我们得到的数据和标签进行训练。

```
from keras import models
from keras import layers
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
```

```
                loss='binary_crossentropy',
                metrics=['acc'])

history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20,
                    validation_data=(validation_features, validation_labe
ls))
```

因为我们只有两个全连接层，所以训练非常快，即使在CPU上一个epoch用时也少于一秒。
接下来我们画出训练过程中的loss和accuracy曲线。

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

image_plot
由图可知，我们达到了90%的验证精度，比我们之前的所有结果都好。同时我们也发现一个问题，虽然我们使用了一个较大比率的dropout，但是还是过拟合了。这是因为这种方法无法采用数据augmentation，对小数据集进行augmentation操作是防止过拟合的一个很必要的操作。
下面我们利用上文提到的第二种方法进行试验，虽然这种方法比较慢并且运行代价很高，但是这种方法可以在训练过程中使用数据augmentation，并且能扩展卷积网络模型，进行端到端的训练。这种方法最好在GPU上进行试验，如果你在CPU上进行试验，最好选用第一种方法。
你可以像添加卷积层一样对Sequential模型添加VGG16卷积网络结构：

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
```

```
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

查看一下模型

`model.summary()`

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_3 (Dense)              (None, 256)               2097408
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
_________________________________________________________________
```

可以发现VGG16的卷及网络结构共有14714688个参数，这是非常大的一个网络。我们添加的网络只有200万个参数。

在我们编译和训练网络之前，我们需要固定住卷积网络结构，固定指的是在训练过程中网络结构的某一层或者某些层参数不会更新，如果我们不固定的话，卷积网络结构在之前的训练集上学习到的特征表示就会发生改变。因为最顶层的全连接层的权值是随机初始化的，所以在网络传播过程中参数会发生很大的改变，这将在很大程度上破坏网络之前学习到的特征表示。

在keras中你可以通过设置一个网络结构的trainable属性为False来固定住网络。

```
print('This is the number of trainable weights '
      'before freezing the conv base:', len(model.trainable_weights))
```

This is the number of trainable weights before freezing the conv base: 30

```
conv_base.trainable = False
print('This is the number of trainable weights '
      'after freezing the conv base:', len(model.trainable_weights))
```

This is the number of trainable weights after freezing the conv base: 4

有了这个设置之后，只有我们添加的全连接层可以被训练，共包括四个tensor：两层，每一层中一个权值矩阵和一个偏置向量。为了让这个设置生效，我们必须先编译这个模型。另外，如果你编译后又改变了可训练权值，那么你还需要重新编译才能生效，否者这些改动就是没有效果的，会被忽略。

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50,
      verbose=2)
model.save('cats_and_dogs_small_3.h5')
```

画出结果

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

imageplot
由图可知，验证集精度达到了96%，这比以前的结果都要好。

Fine-tuning

另一个模型重复使用的方法是fine-tuning，这个是特征提取的补充，fine-tuning由“冰冻”卷积网络结构的部分顶部“解冻”层和我们添加的全连接层构成。称为“微调”是因为这个操作只是稍微调整部分重用结构模型的高层抽象表示，使网络更适合当前工作。
之前我们提到过，为了能够训练一个随机初始化的顶部分类器，我们需要固定住VGG16卷积网络结构。同样的原因，只有顶部的分类器训练好了之后才能对VGG16卷积网络模型的顶层进行fine-tuning。如果没有训练好，那么训练期间传播到网络中的错误信号就会非常大，并且被fine-tuning的层之前学习到的特征表示就会被破坏。因此fine-tuning一个网络的步骤如下：

- 在一个已经训练好的基础网络的顶层添加一个自定义网络结构
- 固定住这个基础网络
- 训练你添加的自定义层网络结构
- 解冻部分基础网络层
- 联合训练这些层和你自定义的部分
  前面特征提取部分，就是我们的前三步，下面进行第四步：解冻整个conv_base，冰冻其中的某些层。
  提醒一下，这就是我们的卷及网络结构：

```
conv_base.summary()
```

我们会微调最后的三个卷积层，也就是说block4_pool层之前的部分都要固定住，block5_conv1, block5_conv2, block5_conv3是可以训练的。
为什么不fine-tuning更多的层？或者fine-tuning整个卷积网络结构？这些我们都可以微调，但是微调之前我们需要考虑一下内容：

- 浅层的卷积网络编码的是更具通用性和更加可重复性的特征，深层的卷积网络编码的是更加特异性的特征。所以微调这些特征对我们更加有用，这是因为这些特征需要重新定义以适应我们新的问题。
- 我们微调的参数越多，那么过拟合的风险就越大。VGG16共有1500万个参数，这么大的模型在我们的小数据集上进行训练是具有很大风险的。
  所以微调个2至3层是一个很好的策略，那就开始吧!

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

现在我们使用RMSprop来微调我们的网络，设置很小的学习率。小学习率可以让我们控制三个微调层的改动幅度不至于过大，从而减少对之前特征表示的损害。
接下来进行编译和训练

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
model.save('cats_and_dogs_small_4.h5')
```

画图

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
```

```
  plt.figure()

  plt.plot(epochs, loss, 'bo', label='Training loss')
  plt.plot(epochs, val_loss, 'b', label='Validation loss')
  plt.title('Training and validation loss')
  plt.legend()

  plt.show()
```