BST 261: Data Science II

Lecture 8

Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

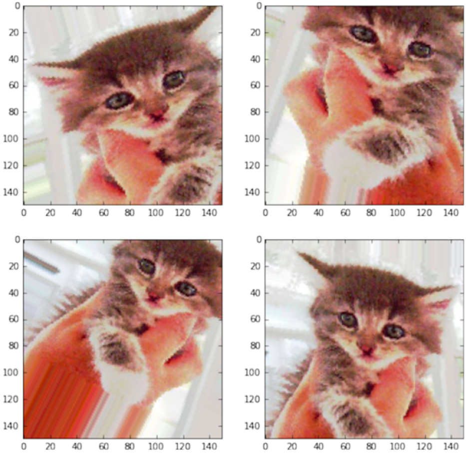
April 11, 2018

Convolutional Neural Networks (CNNs) Continued

Data Augmentation

- As we have seen, overfitting is caused by having too few training examples to learn from
- Data augmentation generates more training data from existing training examples by **augmenting** the samples via a number of random transformations
- These transformations should yield believable images
- Types of augmentation:
    - Rotation
    - Width or height shift
    - Shear
    - Zoom
    - Horizontal flip
    - Fill
- If you train a network using data-augmentation, it will never see the same input twice, but the inputs will still be heavily correlated - you're remixing known information, not producing new information
- May not completely escape overfitting due to this correlation
- Adding dropout can also help

Data Augmentation

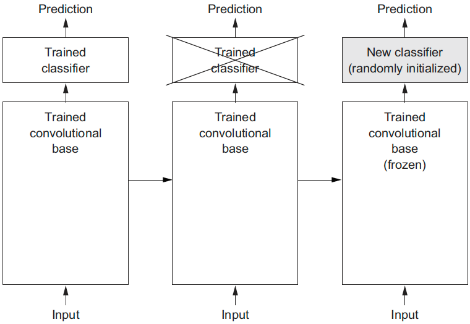


Pretrained Networks

- Another way around having a small number of training examples to learn from, is using networks that have been trained on other, bigger datasets similar to the type of data you have
- A **pretrained network** is a saved network that was previously trained on a large dataset
- If the dataset used to train the network is large enough and big enough, the features learned by the pretrained network can act as a generic model to use as a base for your network
- This saves an enormous amount of computing time
- Pretrained networks can be used for **feature extraction** and **fine-tuning**
- Commonly used pretrained networks include
  - VGG16
  - ResNet
  - Inception
  - Inception-ResNet
  - Xception
- A commonly used dataset used to train a network is the ImageNet dataset
  - 1.4 million labeled images
  - 1,000 different classes
  - Mostly animals and everyday objects

Feature Extraction

- Consists of using the representations learned by a previous network to extract features from new samples
- These features are then run through a new classifier that is trained from scratch, and predictions are made
- For CNNs, the part of the pretrained network you use is called the **convolutional base**, which contains a series of convolution and pooling layers
- For feature extraction, you keep the convolutional base of the pretrained network, remove the dense / trained classifier layers, and append new dense and classifier layers to the convolutional base



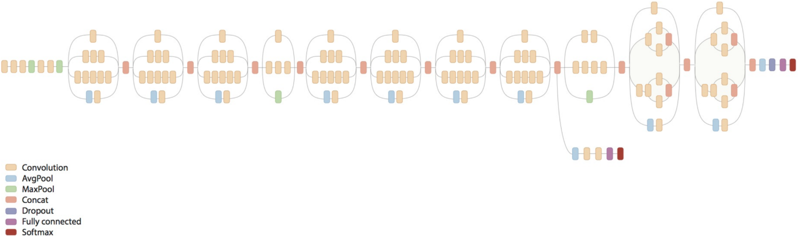
Feature Extraction

- We could also reuse the densely connected classifier as well, but this is not advised
- Representations learned by the convolutional base are likely to be more generic and thus more reusable
- The representations learned by the classifier will be specific to the set on classes the model was trained on
- They will also no longer contain information about where objects are located in the input image
  - This makes them especially useless when the object's location is important
- The level of generality depends on the depth of the layer in the model
  - Early layers extract local, highly generic features, i.e. edges, colors, textures
  - Later layers extract more abstract concepts i.e. "cat ear" or "dog eye"
- If your new dataset is very different from the dataset that was used to train the model, you should use only the first few layers for feature extraction rather than the entire base

Pretrained Networks in Keras

- The following models come prepackaged with Keras:
  - Xception
  - Inception V3
  - ResNet50
  - VGG16
  - VGG19
  - MobileNet

The Inception Model



Instantiating the VGG16 Convolutional Base

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

- `weights`: specify which weight checkpoint to initialize the model from
- `include_top`: refers to including or not the densely-connected classifier on top of the network. By default, this densely-connected classifier would correspond to the 1000 classes from ImageNet.
- `input_shape`: the shape of the image tensors that we will feed to the network. This argument is purely optional: if we don't pass it, then the network will be able to process inputs of any size.

Instantiating the VGG16 Convolutional Base

```
conv_base.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

- The final output has shape $(4, 4, 512)$
- You have 2 options:
  - Feature extraction without augmented data: you can run the convolutional base over the dataset, record its output to a numpy array, and then use these values as input to a densely connected classifier
    - This is fast and cheap to run
    - It won't allow you to use augmented data
  - Feature extraction with augmented data: you can extend the convolutional base by adding dense layers on top and running the whole model on the input data
    - This allows data augmentation
    - This is very computationally expensive

Feature Extraction without Augmented Data

```python
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
```

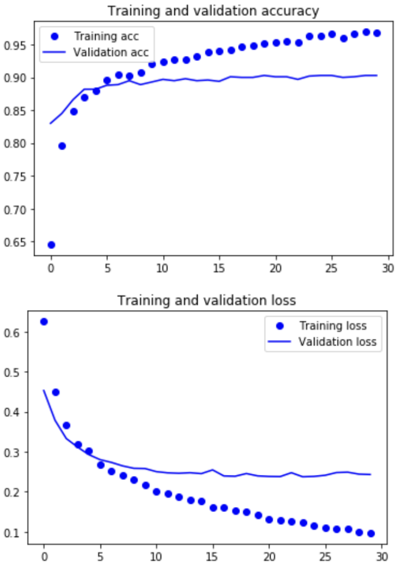
Feature Extraction without Augmented Data

```
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 loop,
            # we must `break` after every image has been seen once.
            break
    return features, labels
```

Feature Extraction without Data Augmentation

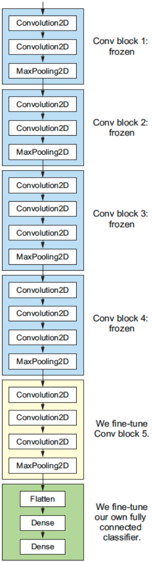
```
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

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))

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_labels))
```

Feature Extraction without Data Augmentation



- You can add the convolutional base just like you add a layer to a network
- It is important that you **freeze** the convolutional base
  - Freezing a layer or set of layers prevents their weights from being updated during training
  - If you do not freeze the base, or part of the base, the representations learned by the pretrained model will be modified and would effectively destroy the representations previously learned and the generalizability and usefulness of the base

Feature Extraction with Data Augmentation



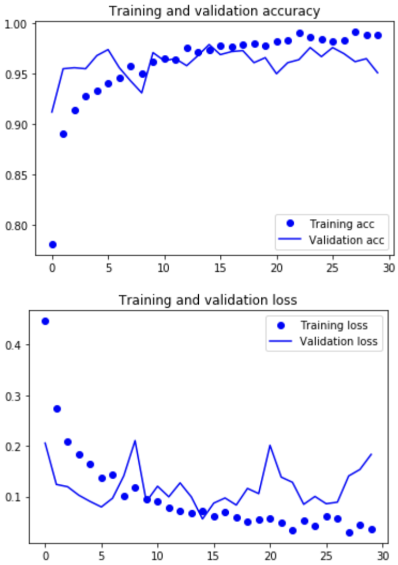
Feature Extraction with Data Augmentation

```
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'))

conv_base.trainable = False

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')
```

Feature Extraction with Data Augmentation

```
# 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)
```

Feature Extraction with Data Augmentation



Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (the dense layers used to classify), and these top unfrozen layers

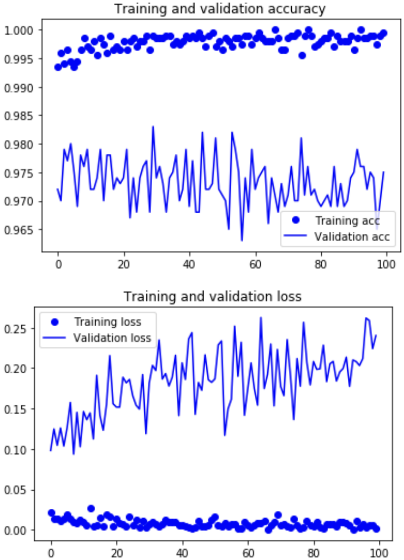
- This slightly adjusts the more abstract representations of the pretrained model in an effort to make them more relevant for the problem at hand
- It is only possible to fine-tune the top layers of the convolutional base, and only after the added classifier layers have been trained
- Steps:
  - Add your custom network on top of an obtained pretrained base network
  - Freeze the base network
  - Train the part you added
  - Unfreeze some layers in the base network
  - Jointly train both the unfrozen layers and top layers

Fine-tuning

- In practice it is good to unfreeze 2-3 top layers of the base
- The more layers you unfreeze, the more parameters that need to be trained, and the higher the risk of overfitting
- Note that earlier layers in the base encode more generic, reusable features, and layers higher up encode more specialized features. Thus, it's more useful to fine-tune layers higher up in the base

Fine-tuning in Keras

```
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
```


Training and validation accuracy
1.000
0.995
0.990
0.985
0.980
0.975
0.970
0.965
0
20
40
60
80
100
Training acc
Validation acc
Training and validation loss
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
Training loss
Validation loss

- It is possible to visualize and interpret the learned representations of your CNN
- 3 of the most useful visualizations are
  - Visualizing intermediate activations
    - Useful for understanding how successive layers transform their input and getting an idea of the meaning of individual filters
  - Visualizing filters
    - Useful for understanding what visual pattern or concept each filter in a CNN is receptive to
  - Visualizing heatmaps of class activations in an image
    - Useful for understanding which parts of an image were identified as belonging to a given class