5.3 Using a pretrained convnet

FAST FEATURE EXTRACTION WITHOUT DATA AUGMENTATION

Listing 5.17 Extracting features using the pretrained convolutional base

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
base_dir = './datasets/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) # WITHOUT DATA AUGMENTATION
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=20,     class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch) # conv_base = VGG16
        features[i * 20 : (i + 1) * 20] = features_batch
        labels[i * 20 : (i + 1) * 20] = labels_batch
        i += 1
        if i * 20 >= sample_count:    # WITHOUT DATA AUGMENTATION
            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)
```

5.3 Using a pretrained convnet

- The extracted **features** are currently of shape `(samples, 4, 4, 512)` → densely connected classifier `(samples, **8192**)`:

flatten

```
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))
```

Listing 5.18 Defining and training the densely connected classifier

```
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))
```

- Training is very fast, because you only have to deal with two `Dense` layers—an epoch takes less than one second even on CPU.

5.3 Using a pretrained convnet

Listing 5.19 Plotting the results

```
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()
```

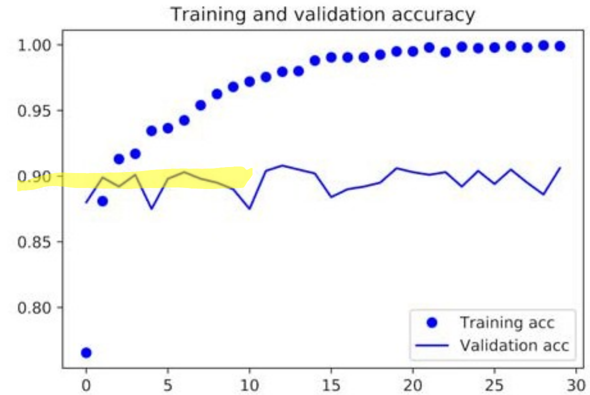


Figure 5.15 Training and validation accuracy for simple feature extraction

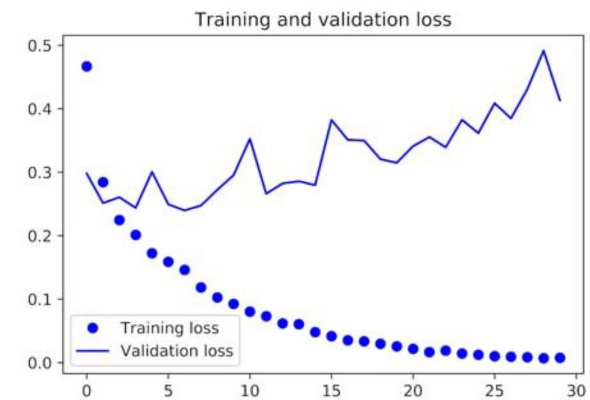


Figure 5.16 Training and validation loss for simple feature extraction

- You reach a validation accuracy of about 90%. But the plots also indicate that you're overfitting almost from the start—despite using dropout with a fairly large rate. That's because this technique doesn't use data augmentation, which is essential for preventing overfitting with small image datasets.

5.3 Using a pretrained convnet

FEATURE EXTRACTION WITH DATA AUGMENTATION

- data augmentation during training - much slower and more expensive
- extending the **conv_base** model and running it end to end on the inputs

NOTE This technique is so expensive that you should only attempt it if you have access to a GPU—it's absolutely intractable on CPU. If you can't run your code on GPU, then the previous technique is the way to go

Listing 5.20 Adding a densely connected classifier on top of the convolutional base

```
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'))
```

```
>>> model.summary()
Layer (type)                     Output Shape          Param #
================================================================
vgg16 (Model)                    (None, 4, 4, 512)     14714688
________________________________________________________________
flatten_1 (Flatten)              (None, 8192)          0
________________________________________________________________
dense_1 (Dense)                  (None, 256)           2097408
________________________________________________________________
dense_2 (Dense)                  (None, 1)             257
================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

5.3 Using a pretrained convnet

FEATURE EXTRACTION WITH DATA AUGMENTATION

- *Freezing* - freeze the convolutional base, preventing weights from being updated during training
- If you don't do this, then the representations that were previously learned by the convolutional base will be modified during training.

- In Keras, you freeze a network by setting its `trainable` attribute to `False`:

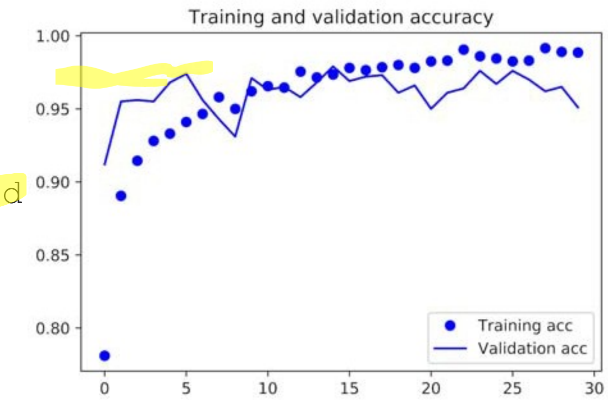
```
>>> print('This is the number of trainable weights'
         'before freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights before freezing the conv base: 30
>>> conv_base.trainable = False
>>> print('This is the number of trainable weights '
         'after freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights after freezing the conv base: 4
```

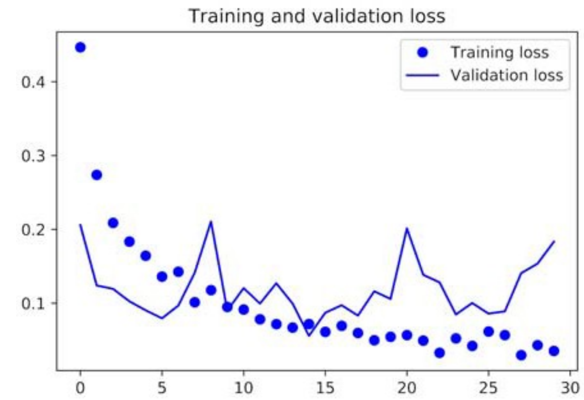
- *2 Dense layers that you added will be trained - That's a total of 4 weight tensors: two per layer (the main weight matrix and the bias vector). (13+2)*2 =30 weight tensors for whole model.*
- *Now you can start training your model, with the same data-augmentation configuration that you used in the previous example.*

5.3 Using a pretrained convnet

Listing 5.21 Training the model end to end with a frozen convolutional base

```
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(
      rescale=1./255,      rotation_range=20,
      width_shift_range=0.1,      height_shift_range=0.1,
      shear_range=0.1,      zoom_range=0.1,
      horizontal_flip=True,      fill_mode='nearest')
test_datagen=ImageDataGenerator(rescale=1./255)#No augmented
train_generator = train_datagen.flow_from_directory(
        train_dir, # 타깃 디렉터리
        target_size=(150, 150), # 150 × 150로 변경
        batch_size=20,
        class_mode='binary') # 이진 레이블
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']) # augmented!
history = model.fit_generator(
        train_generator,  # 2000/100=20 data augmentations
        steps_per_epoch=100,      epochs=30,
        validation_data=validation_generator,
        validation_steps=50,
        verbose=2) # 진행 막대(progress bar)가 나오지 않도록 설정
```





- validation accuracy of 96% - better the small convnet trained from scratch

5.3 Using a pretrained convnet

5.3.2 Fine-tuning

- *fine-tuning* - slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.
- fine-tune the top layers of the convolutional base
- The steps for fine-tuning a network are as follow:
  1. Add your custom network on top of an already-trained base network.
  2. Freeze the base network.
  3. Train the part you added.
  4. Unfreeze some layers in the base network.
  5. Jointly train both these layers and the part you added.

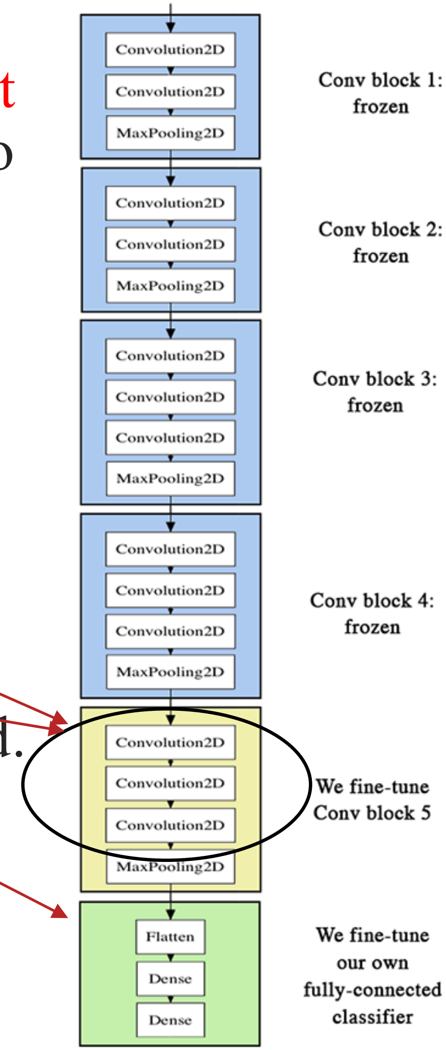


Figure 5.19 Fine-tuning the last convolutional block of the VGG16 network

5.3 Using a pretrained convnet

- fine-tune - the layers `block5_conv1`, `block5_conv2`, and `block5_conv3` should be trainable.
- Why not fine-tune more layers? Why not fine-tune the entire convolutional base? consider the following:
  - Earlier layers - more-generic, reusable features
  - Higher layers - more-specialized features.
  - The more parameters you're training, the more you're at risk of overfitting.
  - The convolutional base has 15 million parameters, so it would be risky to attempt to train it on your small dataset.

Listing 5.22 Freezing all layers up to a specific one

```
conv_base.trainable = True # T,T,…,T
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable: # F,F,…,F,T,T,T
        layer.trainable = True
    else:  # freeze before block5_conv1
        layer.trainable = False
```

```
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: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

freeze

unfreeze

5.3 Using a pretrained convnet

Listing 5.23 Fine-tuning the model

```
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)
```

5.3 Using a pretrained convnet

Listing 5.24 Smoothing the plots

```
def smooth_curve(points, factor=0.8):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
  return smoothed_points

plt.plot(epochs,
         smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs,
         smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs,
         smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs,
         smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

- The validation accuracy curve from about 96% to above 97%
- Note that the loss curve doesn't show any real improvement (in fact, it's deteriorating)

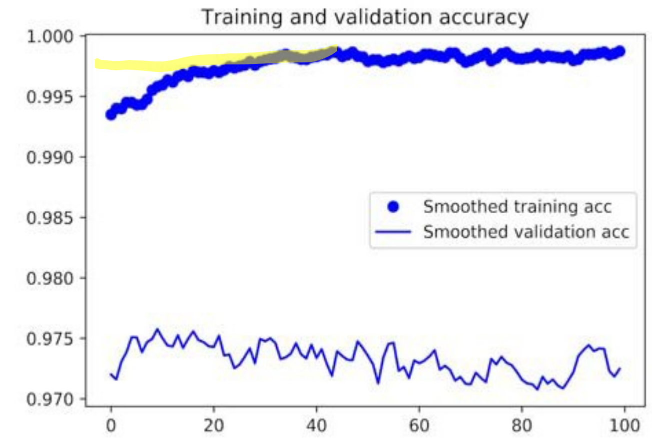


Figure 5.22 Smoothed curves for training and validation accuracy for fine-tuning

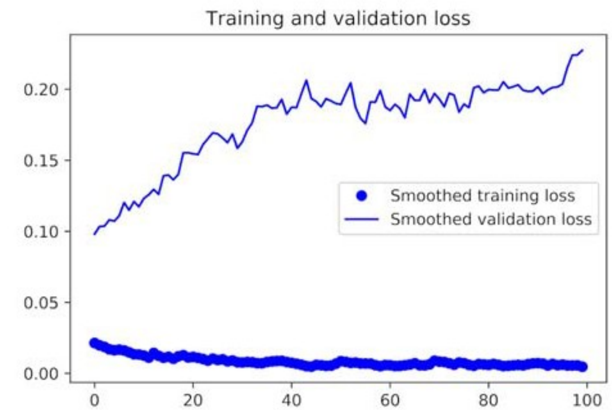


Figure 5.23 Smoothed curves for training and validation loss for fine-tuning

5.3 Using a pretrained convnet

- You can now finally evaluate this model on the test data:

```
test_generator
=test_datagen.flow_from_directory(
     test_dir,
     target_size=(150, 150),
     batch_size=20,
     class_mode='binary')
```

- test accuracy of 92% - In the original Kaggle competition around this dataset, this would have been one of the top results using only a small fraction of the training data available (about 10%). There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!

- 실습

다음 convolution block들을 unfreeze하고 fine tuning 하는 프로그램과 그 결과를 ppt 10 page와 비교하시오.

1) block5_conv3

2) block5_conv3, block5_conv2