

A decorative network diagram in the top-left corner of the slide. It features a complex web of interconnected nodes and edges. The nodes are represented by circles of varying sizes and shades of gray, some with concentric circles. Several nodes are highlighted with solid blue circles, and two are further emphasized with larger blue outlines. The edges are thin, light gray lines connecting the nodes.

# **BST 261: Data Science II**

## **Lectures 5 & 6**

**Convolutional Neural Networks (CNNs): Data  
Augmentation, Pretrained Networks**

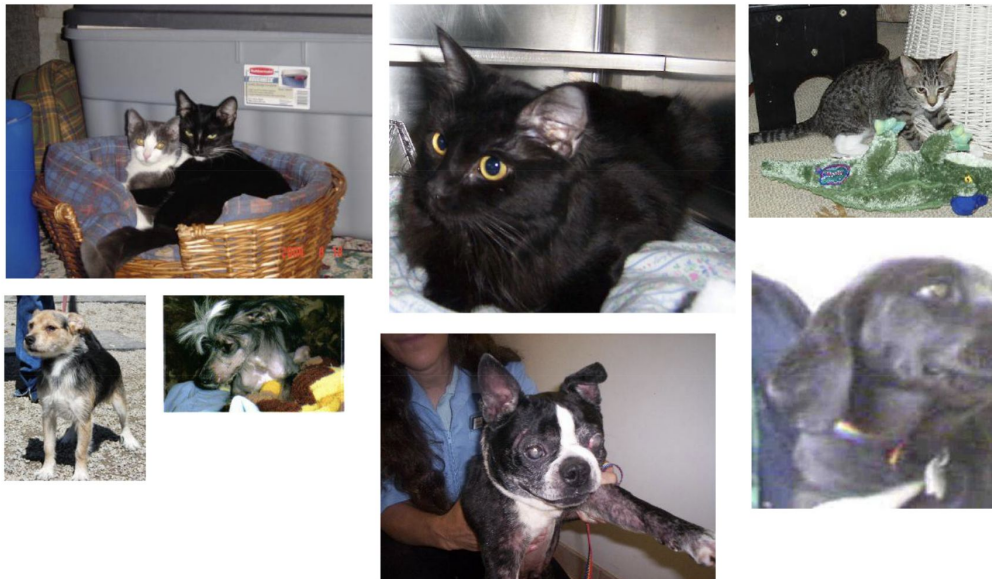
**Heather Mattie**  
**Harvard T.H. Chan School of Public Health**  
**Spring 2 2019**

A decorative network diagram in the bottom-right corner of the slide, mirroring the style of the top-left diagram. It shows a cluster of interconnected nodes and edges. Nodes are represented by circles of varying sizes and shades of gray, with some highlighted by solid blue circles and others by larger blue outlines.

# Classifying Dogs and Cats

The cats vs. dogs dataset that we will use isn't packaged with Keras. It was made available by Kaggle.com as part of a computer vision competition in late 2013, back when convnets weren't quite mainstream. You can download the original dataset [here](#).

- ◎ You will need to create a Kaggle account if you don't already have one
- ◎ The pictures are medium-resolution color JPEGs. They look like this:



# Classifying Dogs and Cats

- ◎ The cats vs. dogs Kaggle competition in 2013 was won by entrants who used CNNs. The best entries could achieve up to 95% accuracy.
- ◎ In our own example, we will get fairly close to this accuracy, even though we will be training our models on less than 10% of the data that was available to the competitors.
- ◎ This original dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543MB large (compressed).
- ◎ After downloading and uncompressing it, we will create a new dataset containing three subsets: a training set with 1000 samples of each class, a validation set with 500 samples of each class, and finally a test set with 500 samples of each class.

```

from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

```

```
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
conv2d_9 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_9 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_10 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_10 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_11 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_11 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_12 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_12 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_3 (Flatten)	(None, 6272)	0
dense_5 (Dense)	(None, 512)	3211776
dense_6 (Dense)	(None, 1)	513
=====		
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		

```

from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

```

Convolution and pooling layers - notice how the output size decreases with each layer. Remember what applying a filter, padding, and/or strides does to an input.



```
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
conv2d_9 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_9 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_10 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_10 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_11 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_11 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_12 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_12 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_3 (Flatten)	(None, 6272)	0
dense_5 (Dense)	(None, 512)	3211776
dense_6 (Dense)	(None, 1)	513
=====		
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		



```

from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

```

Convolution and pooling layers - notice how the output size decreases with each layer. Remember what applying a filter, padding, and/or strides does to an input.

We finish the network with 1 hidden dense layer and 1 output layer. Note that most of the parameters in the model come from the hidden dense layer and not the convolution or pooling layers.

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_9 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_10 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_10 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_11 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_11 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_12 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_12 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_3 (Flatten)	(None, 6272)	0
dense_5 (Dense)	(None, 512)	3211776
dense_6 (Dense)	(None, 1)	513
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		

```

from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

```

Convolution and pooling layers - notice how the output size decreases with each layer. Remember what applying a filter, padding, and/or strides does to an input.

We finish the network with 1 hidden dense layer and 1 output layer. Note that most of the parameters in the model come from the hidden dense layer and not the convolution or pooling layers.

Total # of parameters that need to be learned by the network

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_9 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_10 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_10 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_11 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_11 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_12 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_12 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_3 (Flatten)	(None, 6272)	0
dense_5 (Dense)	(None, 512)	3211776
dense_6 (Dense)	(None, 1)	513
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		

```
from keras import optimizers

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

```
from keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
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')
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

```
for data_batch, labels_batch in train_generator:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break
```

```
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

We first scale the data to get values between 0 and 1.

Then we transform the images to be 150x150 pixels in size (this is arbitrary), declare a batch size of 20 (this is also arbitrary), and declare the class mode (i.e. the type of classification we want to do)

We can see the shape of the images: they are in batches of 20, with each image being represented by 3, 150x150 tensors: one for R, one for G and one for B color “channels”.

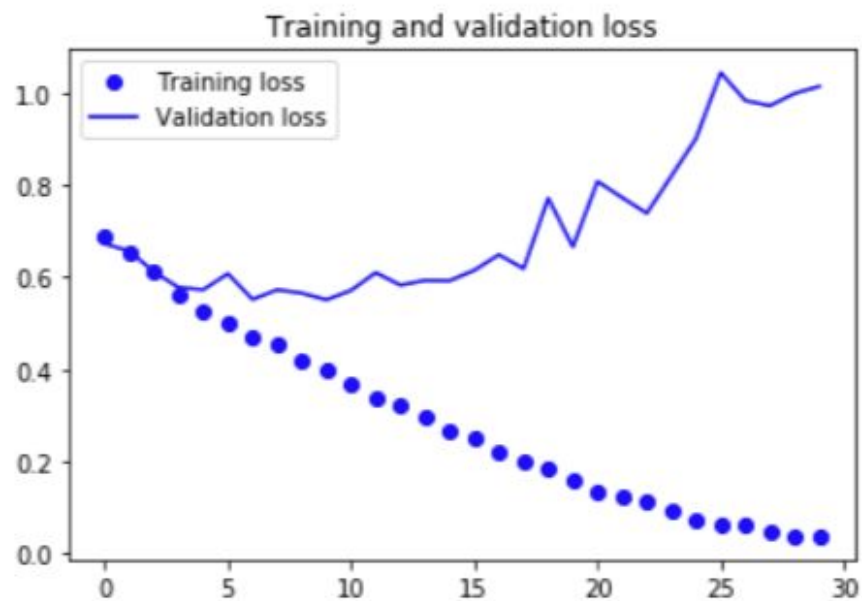
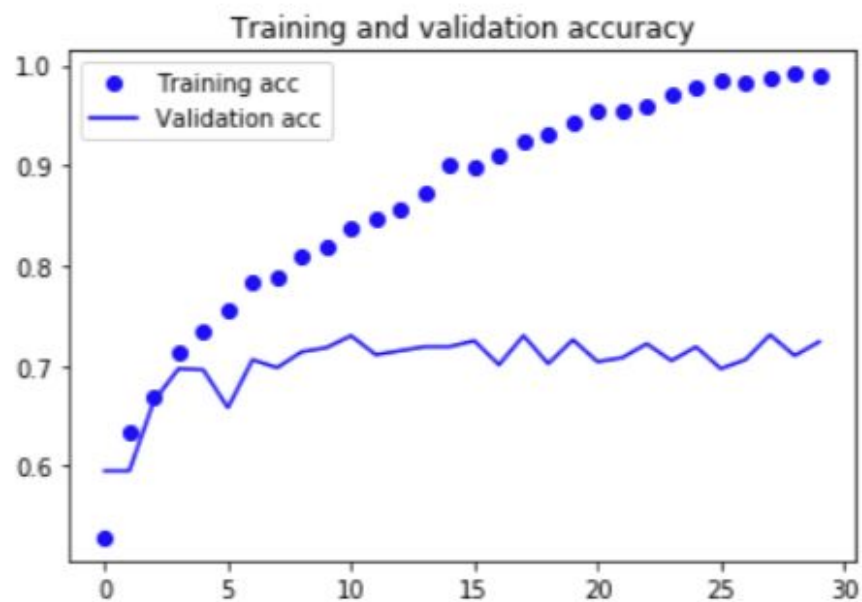


```
history = model.fit_generator(  
    train_generator,  
    steps_per_epoch=100, ←  
    epochs=30,  
    validation_data=validation_generator,  
    validation_steps=50)
```

The number of training examples divided by the batch size, i.e. how many batches we need to go through until the model sees all training data.

```
Epoch 1/30  
100/100 [=====] - 48s 475ms/step - loss: 0.6863 - acc: 0.5475 - val_loss: 0.6995 - val_acc: 0.5200  
Epoch 2/30  
100/100 [=====] - 45s 452ms/step - loss: 0.6579 - acc: 0.6115 - val_loss: 0.6645 - val_acc: 0.5880  
Epoch 3/30  
100/100 [=====] - 46s 455ms/step - loss: 0.6087 - acc: 0.6765 - val_loss: 0.6270 - val_acc: 0.6430  
Epoch 4/30  
100/100 [=====] - 47s 475ms/step - loss: 0.5749 - acc: 0.7045 - val_loss: 0.6011 - val_acc: 0.6610  
Epoch 5/30  
100/100 [=====] - 46s 458ms/step - loss: 0.5436 - acc: 0.7255 - val_loss: 0.5872 - val_acc: 0.6860  
Epoch 6/30  
100/100 [=====] - 44s 440ms/step - loss: 0.5179 - acc: 0.7535 - val_loss: 0.6020 - val_acc: 0.6660  
Epoch 7/30  
100/100 [=====] - 45s 451ms/step - loss: 0.4831 - acc: 0.7630 - val_loss: 0.5909 - val_acc: 0.6840  
Epoch 8/30  
100/100 [=====] - 47s 466ms/step - loss: 0.4620 - acc: 0.7845 - val_loss: 0.5681 - val_acc: 0.6950  
Epoch 9/30  
100/100 [=====] - 45s 450ms/step - loss: 0.4305 - acc: 0.8020 - val_loss: 0.5746 - val_acc: 0.6890  
Epoch 10/30  
100/100 [=====] - 45s 452ms/step - loss: 0.4011 - acc: 0.8110 - val_loss: 0.5733 - val_acc: 0.7020
```

- 
- 
-



# 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
  - Horizontal/vertical flip
  - Random crops/scales
  - ◎ Zoom
  - ◎ Width or height shifts
  - Shearing
  - Brightness, contrast, saturation
  - Lens distortions

# Types of data augmentation

## 1. Rotations





# Types of data augmentation

## 2. Horizontal/Vertical Flips



# Types of data augmentation

## 3. Random crops/scales



# Types of data augmentation

## 4. Shearing





# Types of data augmentation

## 5. Brightness, contrast, saturation





# Types of data augmentation

## 6. Lens distortions



# Types of data augmentation

## 7. Combinations of the above

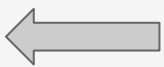


# Data Augmentation

- ◎ 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 in Keras

```
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,)  
  
# 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=32,  
    # 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=32,  
    class_mode='binary')  
  
history = model.fit_generator(  
    train_generator,  
    steps_per_epoch=100,  
    epochs=100,  
    validation_data=validation_generator,  
    validation_steps=50)
```



You can create your own data generator with any specifications you'd like. The values chosen here are arbitrary.

You can check out the [Keras documentation](https://keras.io/guides/data_augmentation/) to see all of the available options.



# Data Augmentation in Keras

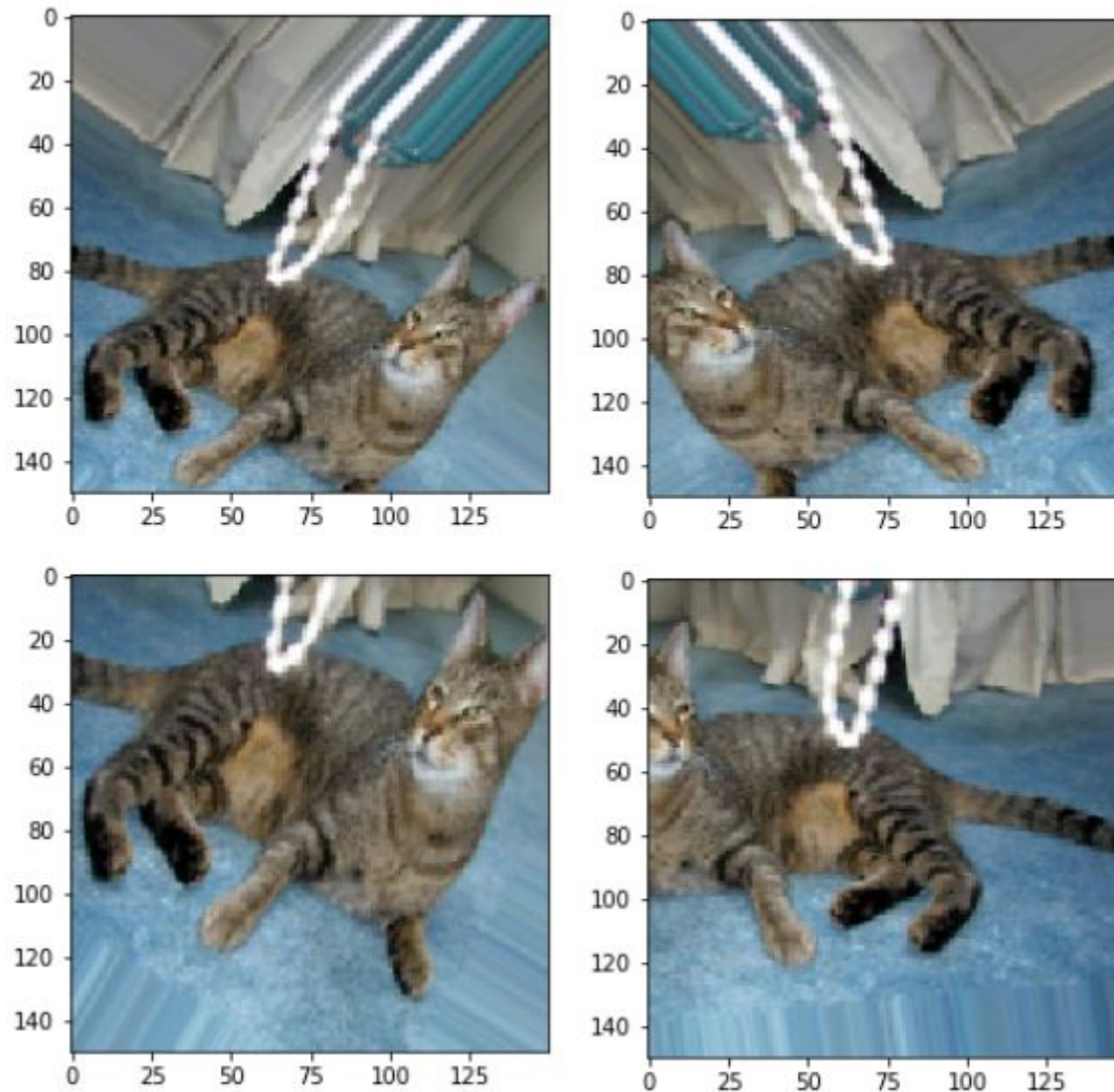
```
train_datagen = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,)  
  
# Note that the validation data should not be augmented!  
test_datagen = ImageDataGenerator(rescale=1./255)  
  
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=32,  
    # 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=32,  
    class_mode='binary')  
  
history = model.fit_generator(  
    train_generator,  
    steps_per_epoch=100,  
    epochs=100,  
    validation_data=validation_generator,  
    validation_steps=50)
```

You can create your own data generator with any specifications you'd like. The values chosen here are arbitrary.

You can check out the [Keras documentation](#) to see all of the available options.

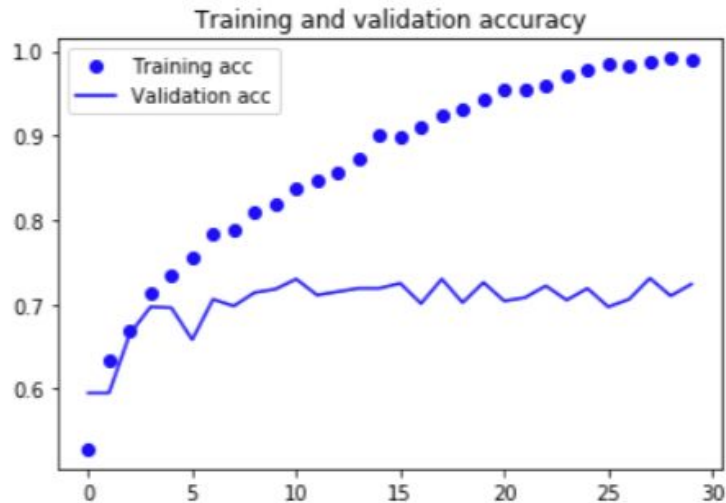
Only the training data should be augmented - not the test or validation sets. The point of augmentation is to “increase” your training set size.

# Data Augmentation in Keras

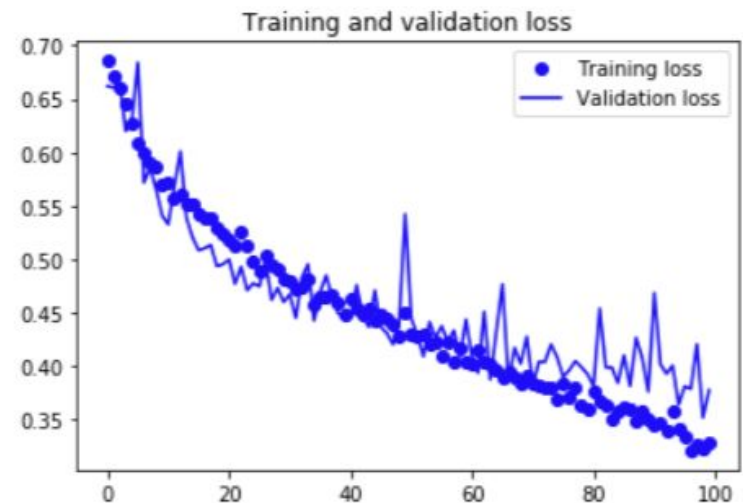
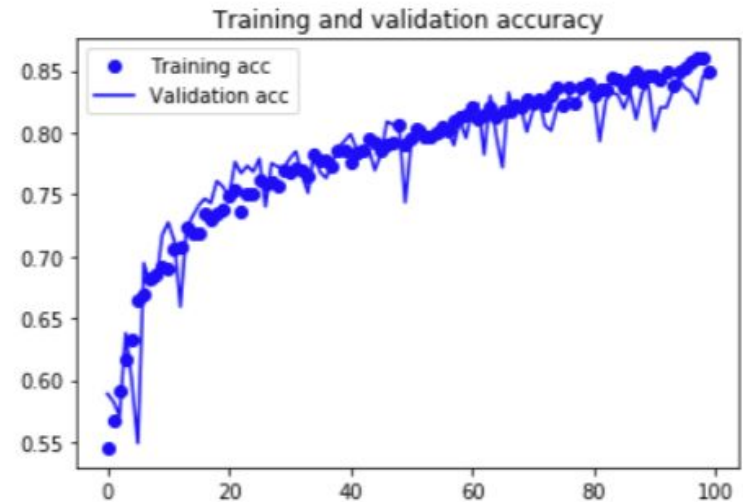


# Data Augmentation in Keras

Without augmentation



With augmentation



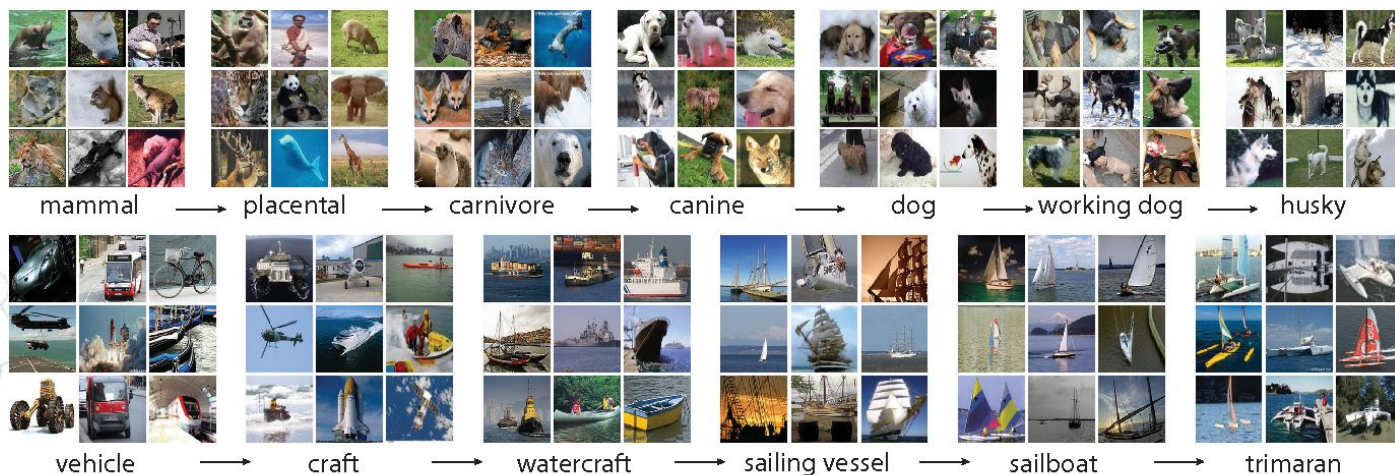
# Pretrained Networks

- ◎ 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**



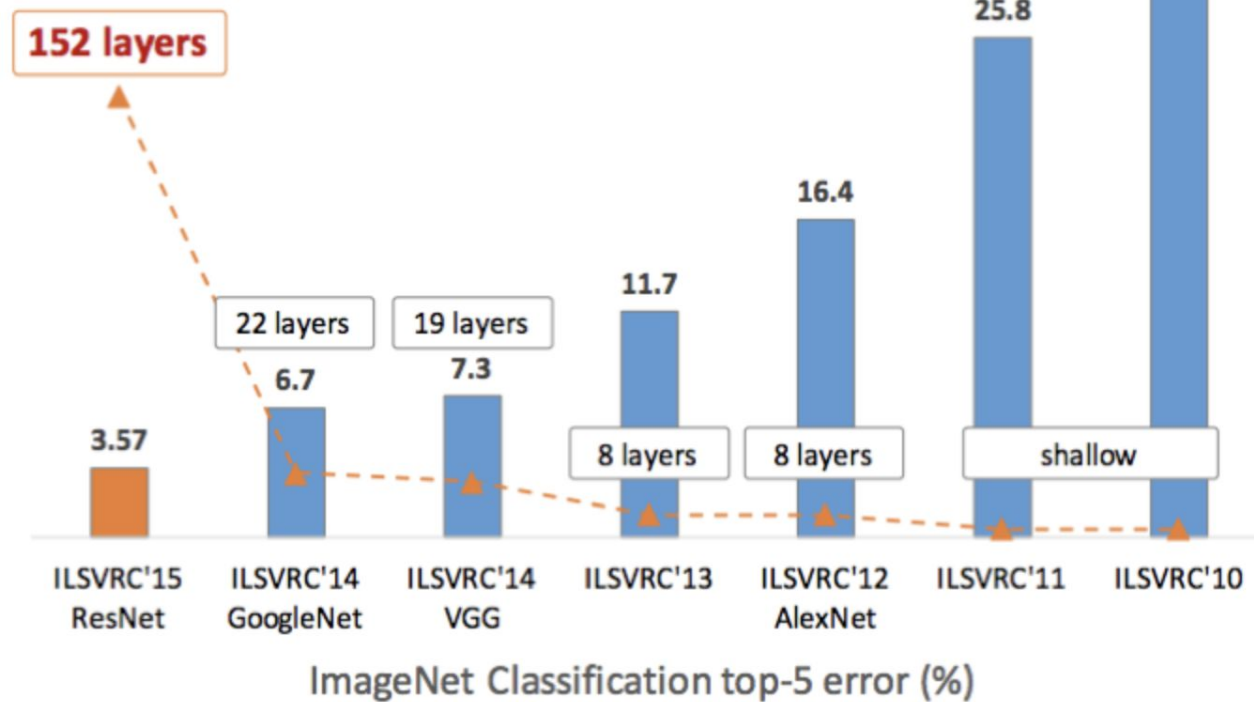
# Pretrained Networks

- Commonly used pretrained networks include
  - VGG16
  - ResNet
  - Inception
  - Inception-ResNet
  - Xception
- 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

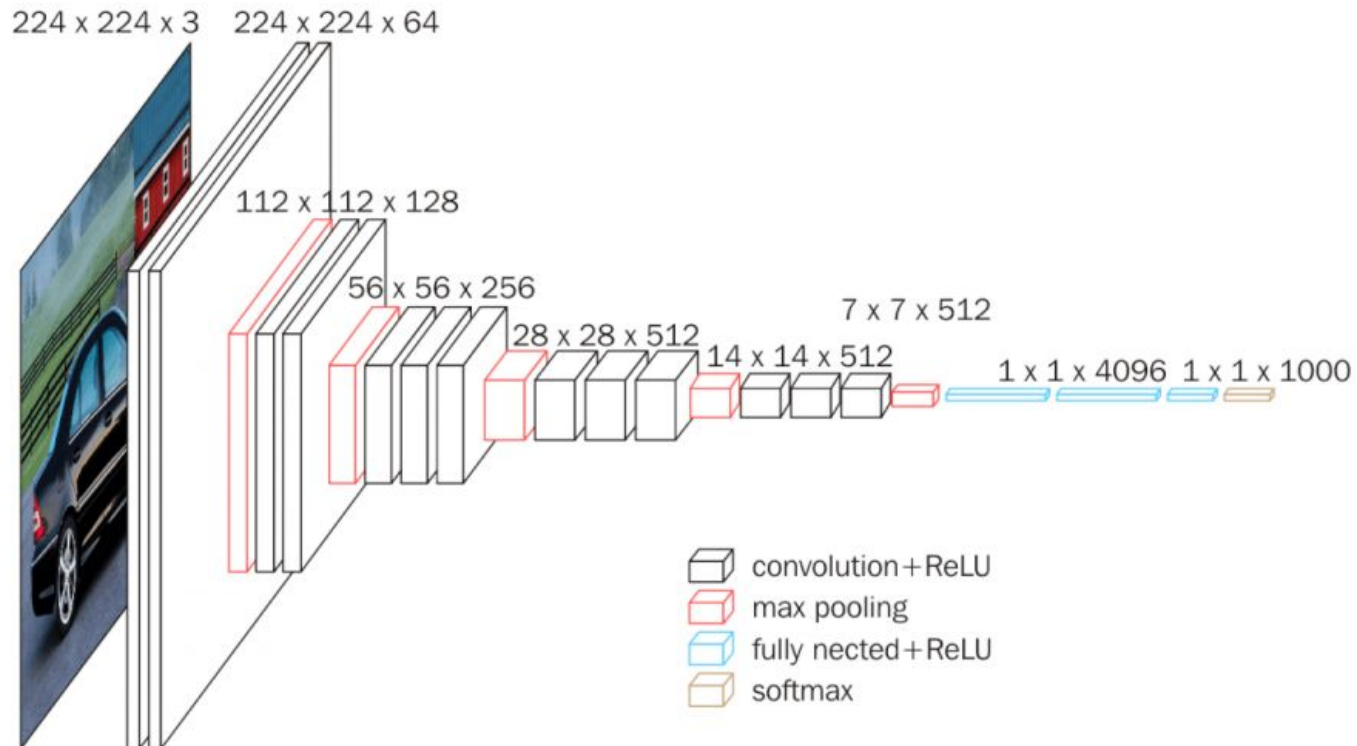
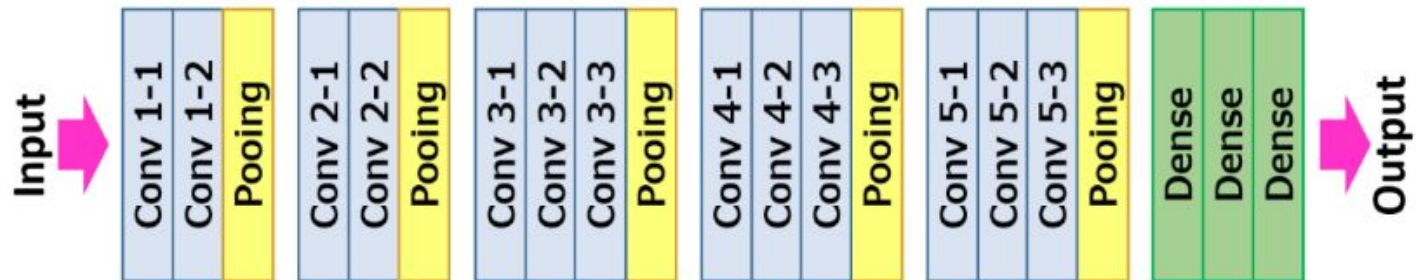


# Pretrained Networks

## Revolution of Depth

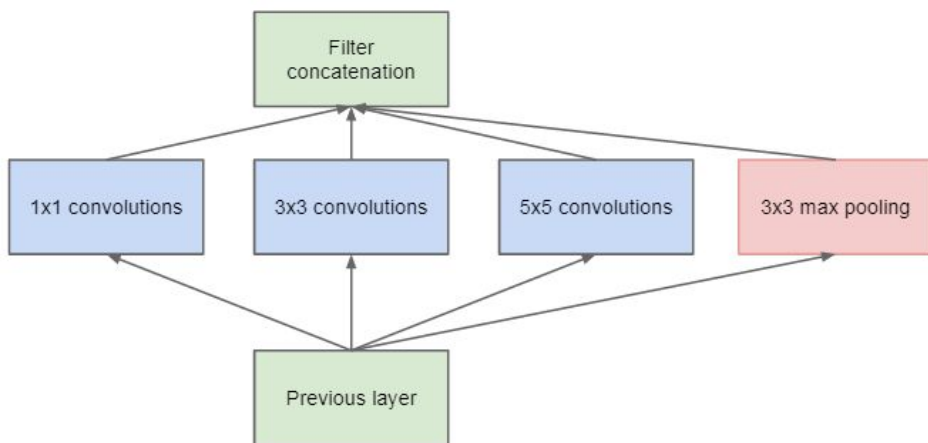


## VGG-16

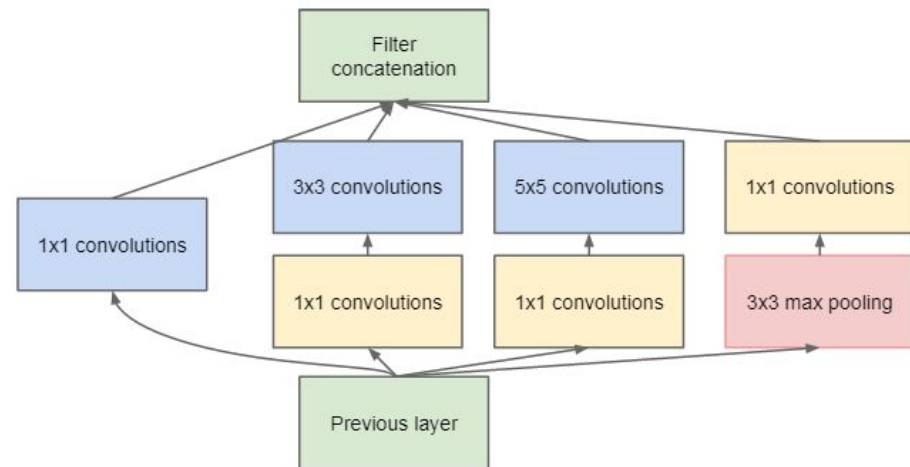




# Inception Models



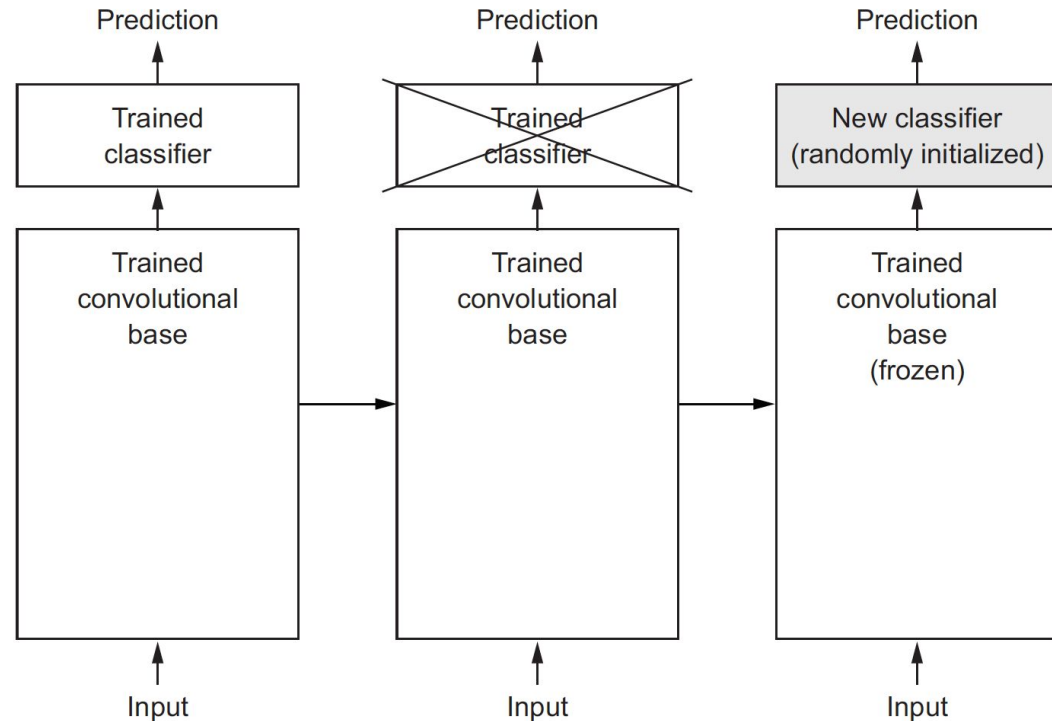
(a) Inception module, naïve version



(b) Inception module with dimension reductions

# 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 of 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

- ⊙ Xception
- ⊙ Inception V3
- ⊙ ResNet50
- ⊙ VGG16
- ⊙ VGG19
- ⊙ MobileNet

# Instantiating the VGG16 Base

```
from keras.applications import VGG16

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

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		

# Instantiating the VGG16 Base

```
from keras.applications import VGG16

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

The final layer is a pooling layer and the final output shape from this base is (4, 4, 512). We need this information when adding layers to the 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

# Instantiating the VGG16 Base

```
from keras.applications import VGG16

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

Note how many more parameters are needed for this model than the previous model we fit.

We'll talk about 2 ways to use a convolutional base, but running models with this many parameters must be done using GPUs.

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		



# Using a Pretrained Network

- ◎ 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

# Feature Extraction without Augmented Data

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

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 loop,
            # 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)
```

# Feature Extraction without Augmented Data

```
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'])
```

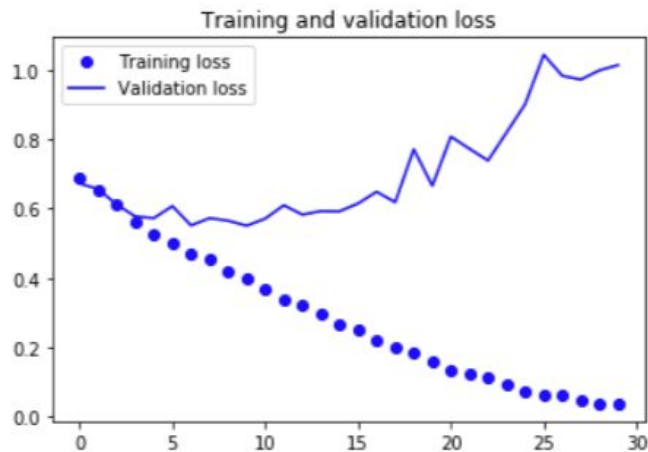
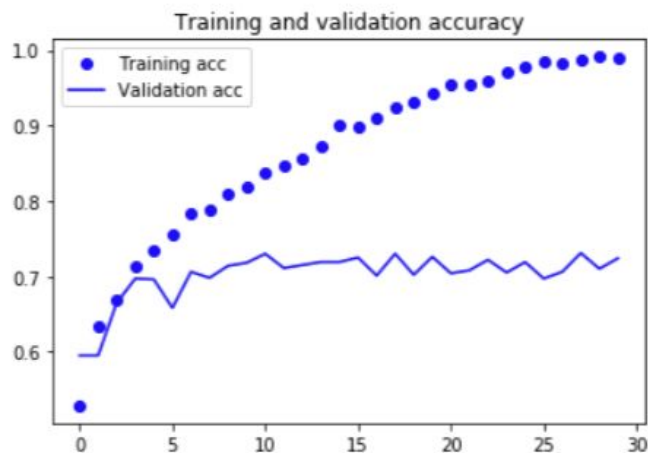
```
history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20,
                    validation_data=(validation_features, validation_labels))
```

We need to reshape the outputs so we can feed them into a dense layer - recall that dense layers take in vectors.

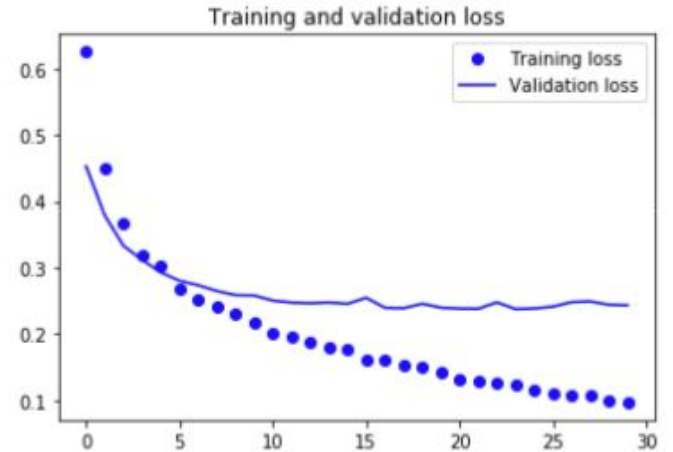
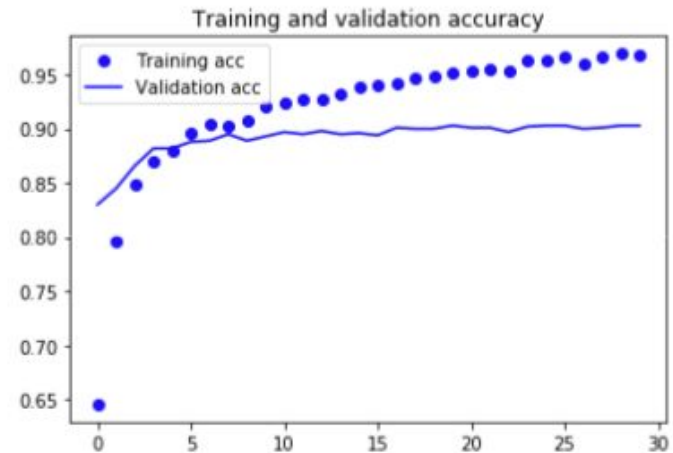


# Feature Extraction without Augmented Data

Original CNN made from scratch



CNN using pretrained base



The background of the slide is a light gray network diagram. It consists of numerous small circular nodes, some of which are solid gray and others are hollow with a gray outline. These nodes are interconnected by a web of thin, light gray lines, creating a complex, interconnected pattern that resembles a neural network or a data graph.

# Feature Extraction with Augmented Data

# Feature Extraction with Augmented Data

```
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		

```
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'])

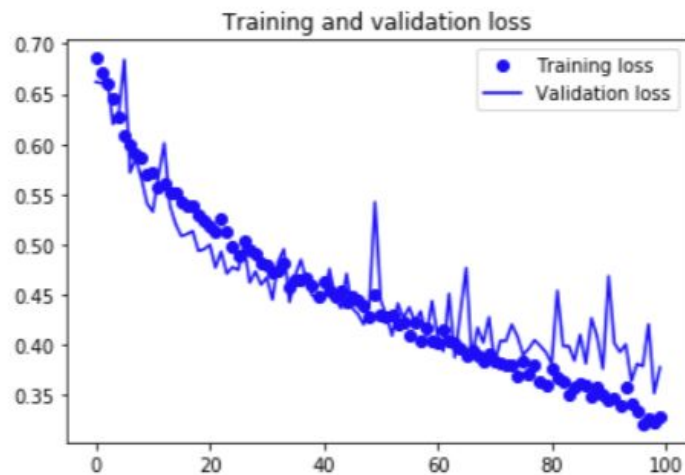
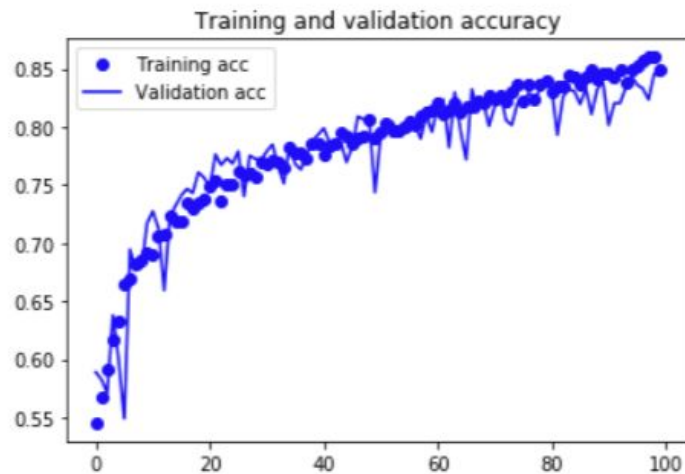
history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=30,
    validation_data=validation_generator,
    validation_steps=50,
    verbose=2)
```

Note: do not run this code without access to a GPU.

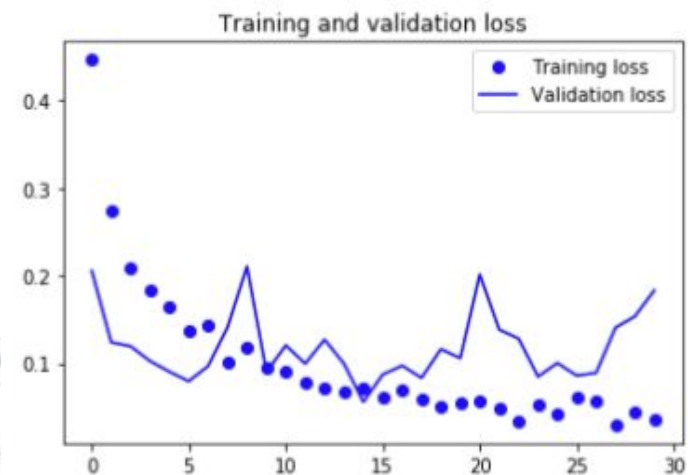


# Feature Extraction with Augmented Data

Original CNN made from scratch  
with data augmentation



CNN using pretrained base  
with data augmentation



The background of the slide features a complex, repeating pattern of a network graph. It consists of numerous small, light-blue circular nodes, some of which are solid and others are hollow with a dashed outline. These nodes are interconnected by a web of thin, light-blue lines, creating a dense, interconnected mesh that covers the entire slide area.

# Fine-tuning

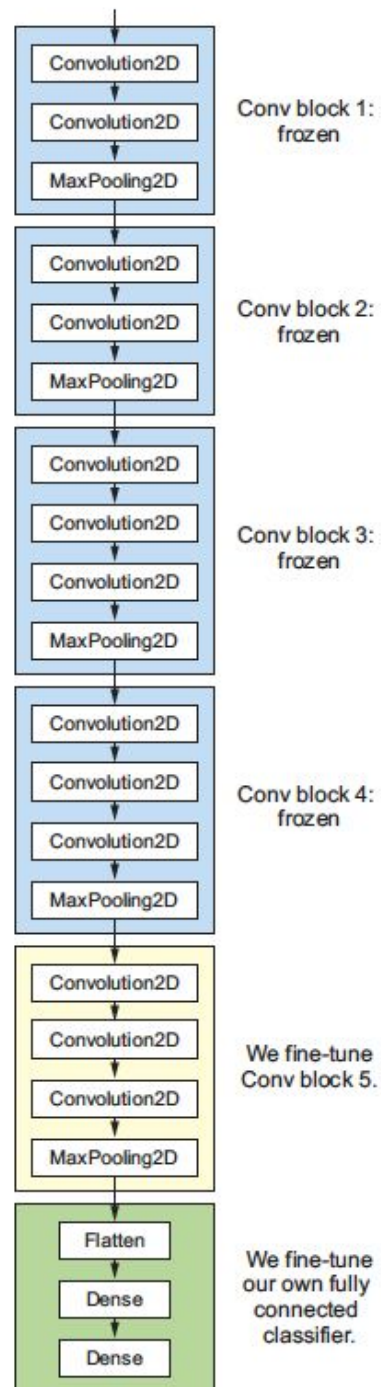
# Fine-tuning

- ◎ **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





We only unfreeze and fine-tune the last block of layers.

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

# 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
```

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

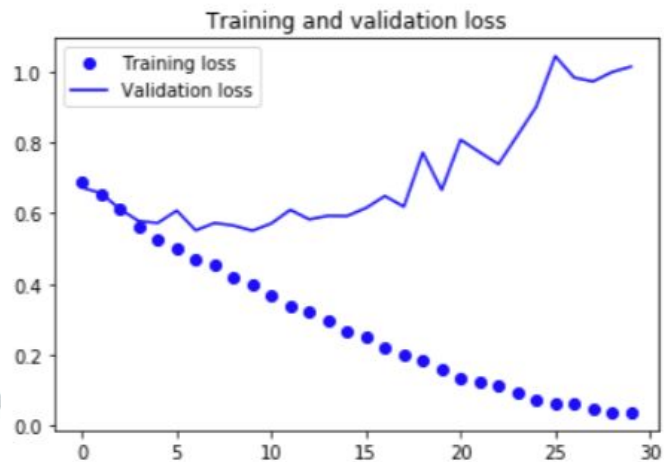
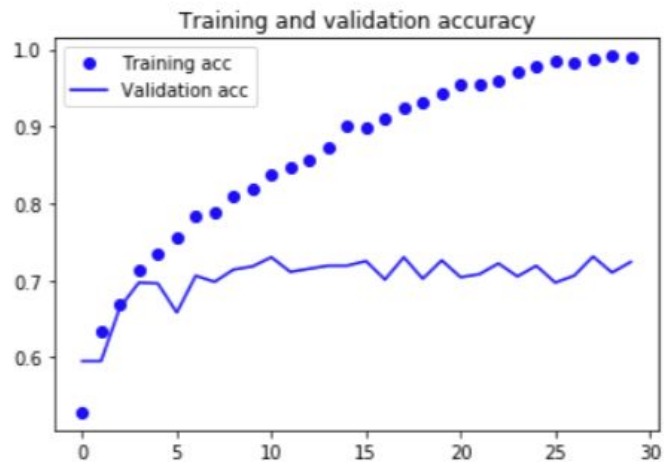
history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=50)
```



We need to say which pretrained blocks should be kept frozen (untrainable) and which one we want to unfreeze (trainable).

# Fine-tuning in Keras

Original CNN made from scratch



CNN made from fine-tuning convolutional base

