Implement a CNN Using Keras

1. Load and Process the MNIST Dataset

Load data

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

print('Shape of x_train: ' + str(x_train.shape))
print('Shape of x_test: ' + str(x_test.shape))
print('Shape of y_train: ' + str(y_train.shape))
print('Shape of y_test: ' + str(y_test.shape))
```

```
Shape of x_train: (60000, 28, 28)
Shape of x_test: (10000, 28, 28)
Shape of y_train: (60000,)
Shape of y_test: (10000,)
```

1. Load and Process the MNIST Dataset

Reshape: convert the $60000 \times 28 \times 28$ dataset to a $60000 \times 28 \times 28 \times 1$ tensor.

```
x_train_vec = x_train.reshape((60000, 28, 28, 1)) / 255.0
x_test_vec = x_test.reshape((10000, 28, 28, 1)) / 255.0

print('Shape of x_train_vec is ' + str(x_train_vec.shape))
```

```
Shape of x_train_vec is (60000, 28, 28, 1)
```

1. Load and Process the MNIST Dataset

One-hot encode: convert the labels (scalars in $\{0, 1, \cdots, 9\}$) to 10-dim vectors.

```
def to_one_hot(labels, dimension=10):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

y_train_vec = to_one_hot(y_train)
y_test_vec = to_one_hot(y_test)
print('Shape of y_train_vec is ' + str(y_train_vec.shape))
```

```
Shape of y_train_vec is (60000, 10)
```

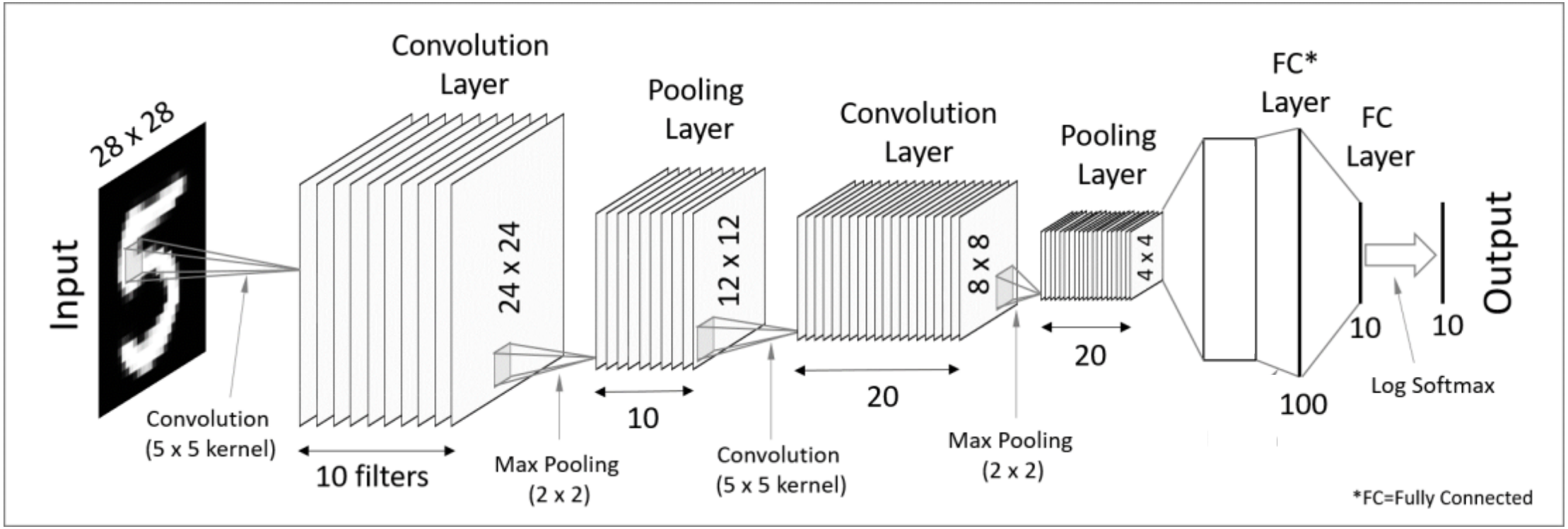
1. Load and Process the MNIST Dataset

Partition the **training** set to **training** and **validation** sets.

```
# Partition to training and validation sets

rand_indices = np.random.permutation(60000)
train_indices = rand_indices[0:50000]
valid_indices = rand_indices[50000:60000]

x_valid_vec = x_train_vec[valid_indices, :, :, :]
y_valid_vec = y_train_vec[valid_indices, :]

x_train_vec = x_train_vec[train_indices, :, :, :]
y_train_vec = y_train_vec[train_indices, :]

print('Shape of x_valid_vec: ' + str(x_valid_vec.shape))
print('Shape of y_valid_vec: ' + str(y_valid_vec.shape))
print('Shape of x_train_vec: ' + str(x_train_vec.shape))
print('Shape of y_train_vec: ' + str(y_train_vec.shape))
```

```
Shape of x_valid_vec: (10000, 28, 28, 1)
Shape of y_valid_vec: (10000, 10)
Shape of x_train_vec: (50000, 28, 28, 1)
Shape of y_train_vec: (50000, 10)
```

2. Build the CNN



2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
```

Filter number

Filter shape

Default

- Stride: 1
- Zero-padding: no padding

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
```

pool_size

Default

- Stride: pool_size

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))  Output: 12×12×10
model.add(layers.Conv2D(20, (5, 5), activation='relu'))  Output: 8×8×20
model.add(layers.MaxPooling2D((2, 2)))  Output: 4×4×20
```

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))  Output: 12×12×10
model.add(layers.Conv2D(20, (5, 5), activation='relu'))  Output: 8×8×20
model.add(layers.MaxPooling2D((2, 2)))  Output: 4×4×20
model.add(layers.Flatten())             Output: 320
```

2. Build the CNN

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(10, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(20, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Output: 12×12×10

Output: 8×8×20

Output: 4×4×20

Output: 320

Output: 100

Output: 10

2. Build the CNN

```
# print the summary of the model.
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 10)        260
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 10)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 20)          5020
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 20)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 320)               0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               32100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 38,390
Trainable params: 38,390
Non-trainable params: 0
_________________________________________________________________
```

3. Train the CNN

Specify: optimization algorithm, learning rate (LR), loss function, and metric.

```
from keras import optimizers
model.compile(optimizers.RMSprop(lr=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

3. Train the CNN

Specify: batch size and number of epochs.

```
history = model.fit(x_train_vec, y_train_vec,
                    batch_size=128, epochs=50,
                    validation_data=(x_valid_vec, y_valid_vec))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/50
50000/50000 [==============================] - 12s 239us/step - loss: 1.2455 - acc: 0.7201 - val_loss: 0.4805 - val_acc: 0.8707
Epoch 2/50
50000/50000 [==============================] - 12s 235us/step - loss: 0.3563 - acc: 0.8995 - val_loss: 0.2851 - val_acc: 0.9172
Epoch 3/50
50000/50000 [==============================] - 12s 244us/step - loss: 0.2474 - acc: 0.9271 - val_loss: 0.2230 - val_acc: 0.9365
```

- ●
- ●
- ●

```
Epoch 49/50
50000/50000 [==============================] - 12s 240us/step - loss: 0.0244 - acc: 0.9924 - val_loss: 0.0439 - val_acc: 0.9875
Epoch 50/50
50000/50000 [==============================] - 12s 239us/step - loss: 0.0238 - acc: 0.9927 - val_loss: 0.0446 - val_acc: 0.9881
```

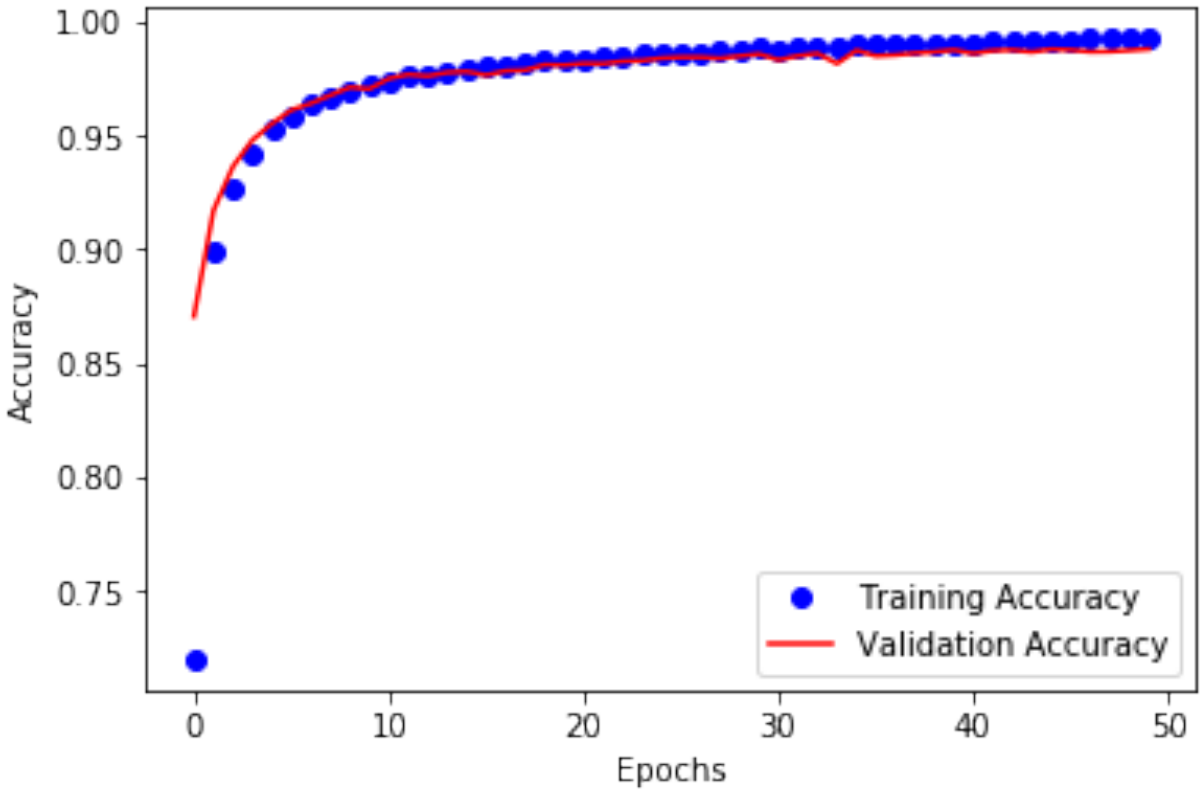
4. Examine the Results

Plot the ***accuracy*** against ***epochs*** (1 epoch = 1 pass over the data).

```
import matplotlib.pyplot as plt
%matplotlib inline

epochs = range(50) # 50 is the number of epochs
train_acc = history.history['acc']
valid_acc = history.history['val_acc']
plt.plot(epochs, train_acc, 'bo', label='Training Accuracy')
plt.plot(epochs, valid_acc, 'r', label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

4. Examine the Results

Plot the ***accuracy*** against ***epochs*** (1 epoch = 1 pass over the data).



Why using the validation set?

- Tune the hyper-parameters, e.g., learning rate, filter shape, filter number, etc.
- Check overfitting or underfitting.

4. Examine the Results

Evaluate the model on the ***test*** set. (So far, the model has not seen the ***test*** set.)

```
loss_and_acc = model.evaluate(x_test_vec, y_test_vec)
print('loss = ' + str(loss_and_acc[0]))
print('accuracy = ' + str(loss_and_acc[1]))
```

```
10000/10000 [==============================] - 1s 127us/step
loss = 0.035708001047297 15
accuracy = 0.9883
```

4. Examine the Results

- Training accuracy: 99.3%
- Validation accuracy: 98.8%
- Test accuracy: 98.8%

Summary

1. Convolution

- Matrix convolutional and tensor convolution.
- Concepts:
  - filter (kernel)
  - patch
  - output (feature map)
  - zero-padding
  - stride

2. Convolutional Neural Network

- Convolutional layers.
  - Filter number.
  - Filter shape.
  - Stride (default: one).
  - Zero-padding (default: no padding).

2. Convolutional Neural Network

- Convolutional layers.
  - Filter number.
  - Filter shape.
  - Stride (default: one).
  - Zero-padding (default: no padding).
- Activation functions.
- Pooling layers.
  - MaxPool, AveragePool, etc.
  - Pool size.
  - Pool stride (default: pool size).

3. Build and Train CNN using Keras

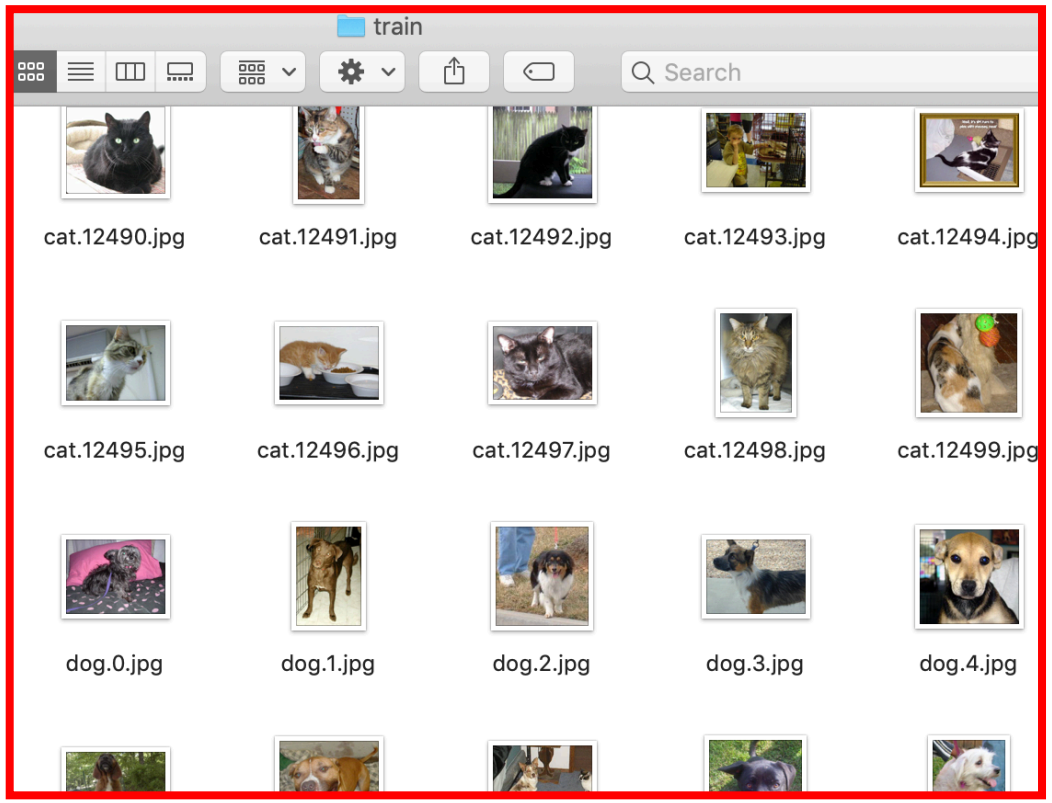
- Build a network.
  - Add Conv, Pool, FC layers.
- Compile the model.
  - Specify optimization algorithm, loss function, and metrics.
- Fit the model on training data.

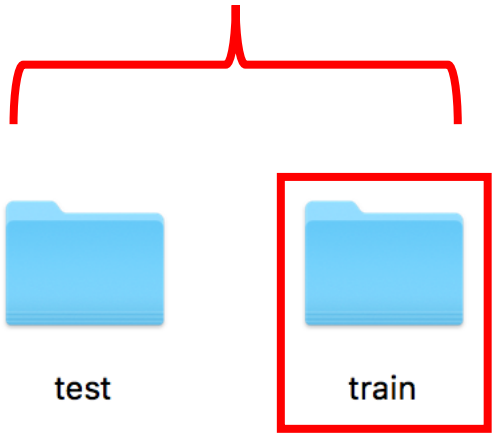
The Dogs vs. Cats Dataset

The Dogs vs. Cats Dataset

The Dogs vs. Cats Dataset

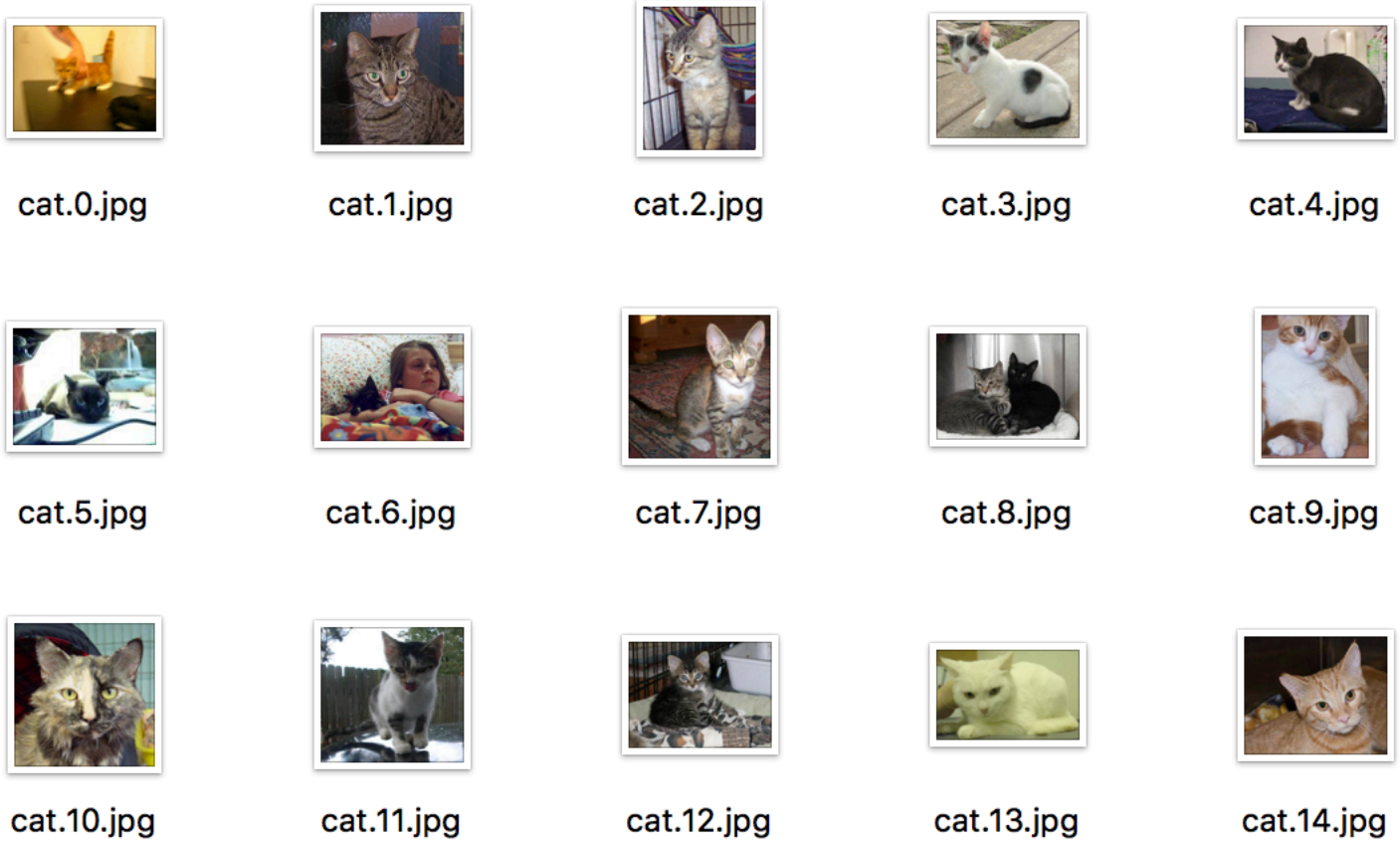
Download: https://www.kaggle.com/c/dogs-vs-cats/data

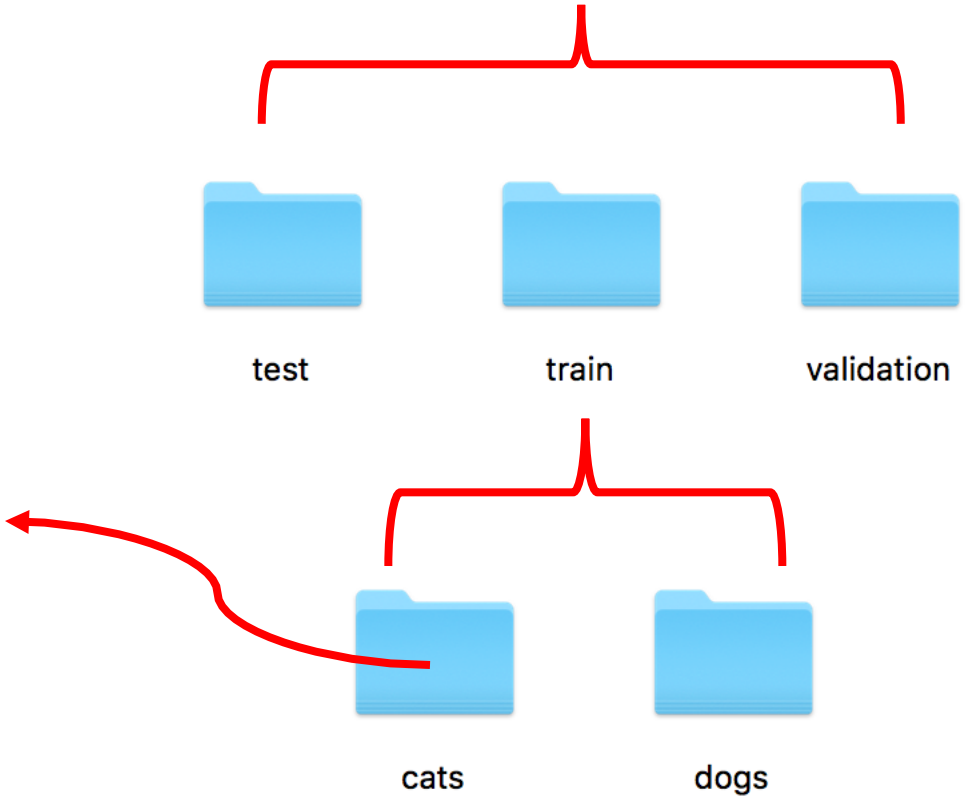




The Dogs vs. Cats Dataset

Download: https://www.kaggle.com/c/dogs-vs-cats/data





The Dogs vs. Cats Dataset

- The dataset has 25,000 training samples.

- I use a subset:
  - 2000 images for training,
  - 1000 images for validation,
  - 1000 images for test.

Implement a CNN Using Keras

1. Load and Process the Dataset

- Currently, the data sit on a drive as JPEG files.
- Data processing:
  1. Read the picture files.
  2. Decode the JPEG content to order-3 tensors.
  3. Resize the images to the the same shape, e.g., $150 \times 150 \times 3$.
  4. Rescale the pixel values (between 0 and 255) to the [0, 1] interval.

1. Load and Process the Dataset

```
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)
```

1. Load and Process the Dataset

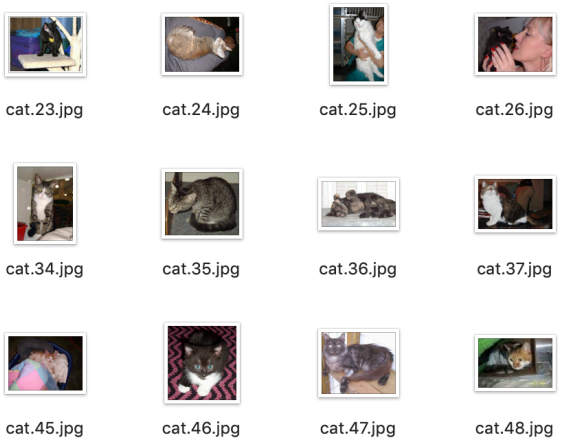
```
from keras.preprocessing.image import ImageDataGenerator

# 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')
```

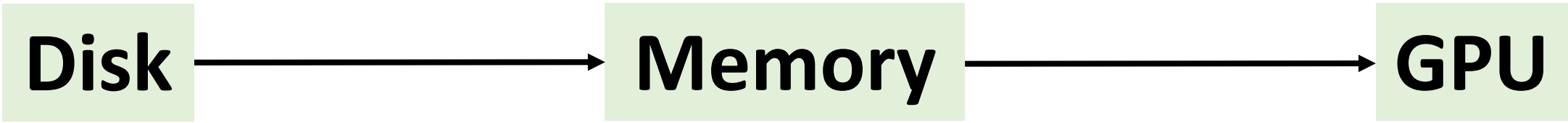
1. Load and Process the Dataset

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

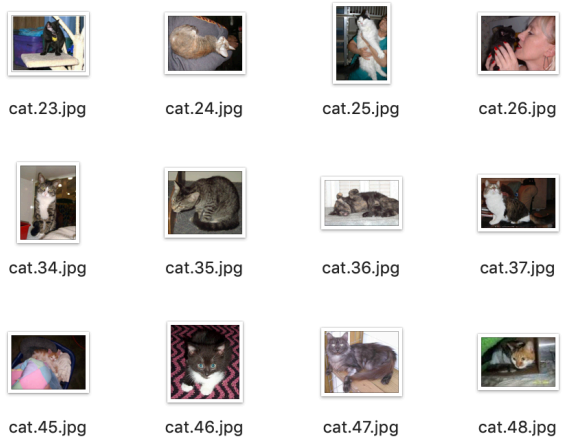
```
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

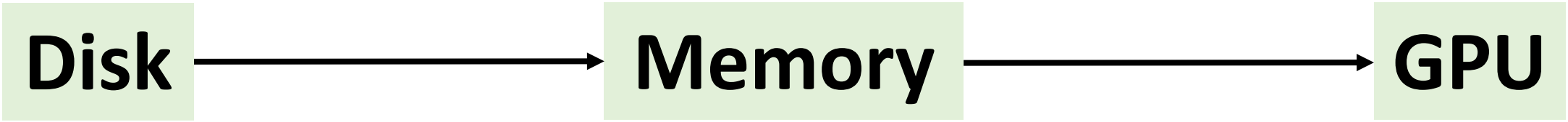
1. Load and Process the Dataset





1. Load and Process the Dataset





2. Build the CNN

```
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'))
```

2. Build the CNN

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

3. Train the CNN

Specify: optimization method, learning rate (LR), loss function, and metric.

```
from keras import optimizers

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

3. Train the CNN

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

3. Train the CNN

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

- Totally $n = 2000$ training samples.
- Batch size is $b = 20$.
- Thus $\frac{n}{b} = 100$ batches per epoch.

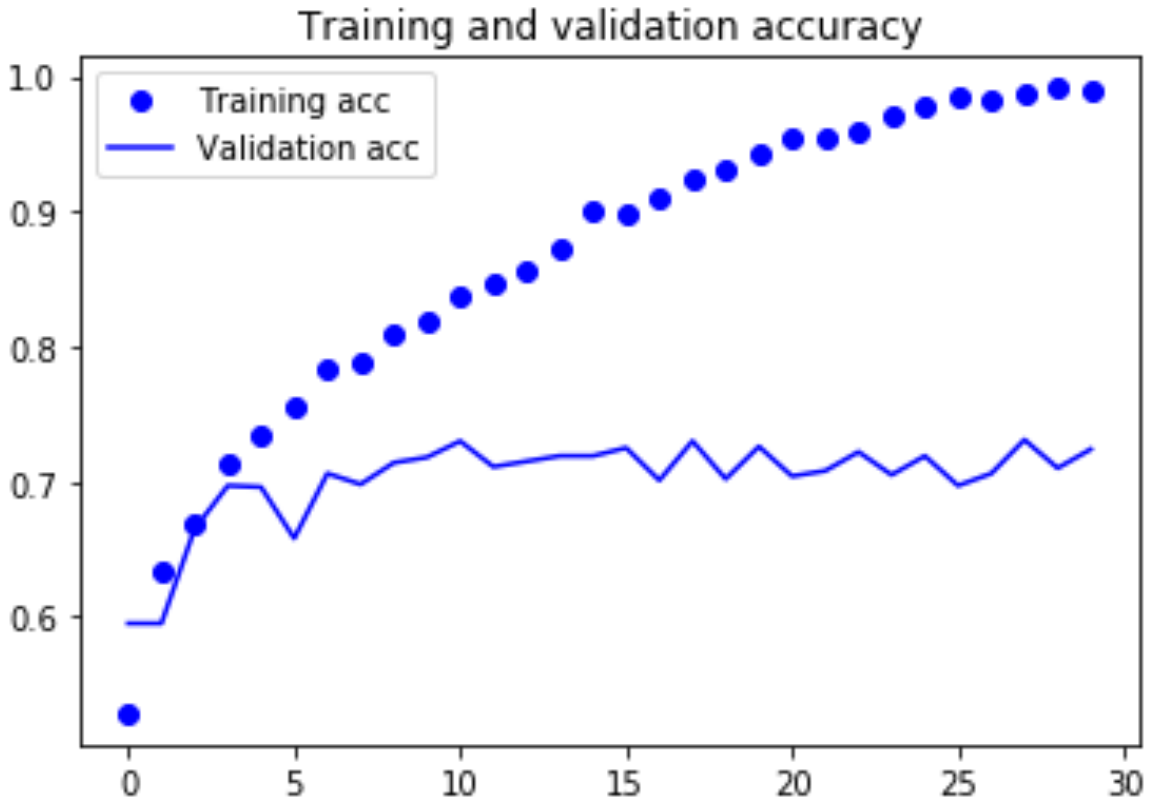
```
Epoch 1/30
100/100 [==============================] - 9s - loss: 0.6898 - acc: 0.5285 - val_loss: 0.6724 - val_acc: 0.5950
Epoch 2/30
100/100 [==============================] - 8s - loss: 0.6543 - acc: 0.6340 - val_loss: 0.6565 - val_acc: 0.5950
Epoch 3/30
100/100 [==============================] - 8s - loss: 0.6143 - acc: 0.6690 - val_loss: 0.6116 - val_acc: 0.6650
Epoch 4/30
100/100 [==============================] - 8s - loss: 0.5626 - acc: 0.7125 - val_loss: 0.5774 - val_acc: 0.6970
```

-
-
-

```
Epoch 29/30
100/100 [==============================] - 8s - loss: 0.0375 - acc: 0.9915 - val_loss: 0.9987 - val_acc: 0.7100
Epoch 30/30
100/100 [==============================] - 8s - loss: 0.0387 - acc: 0.9895 - val_loss: 1.0139 - val_acc: 0.7240
```

4. Examine the Results

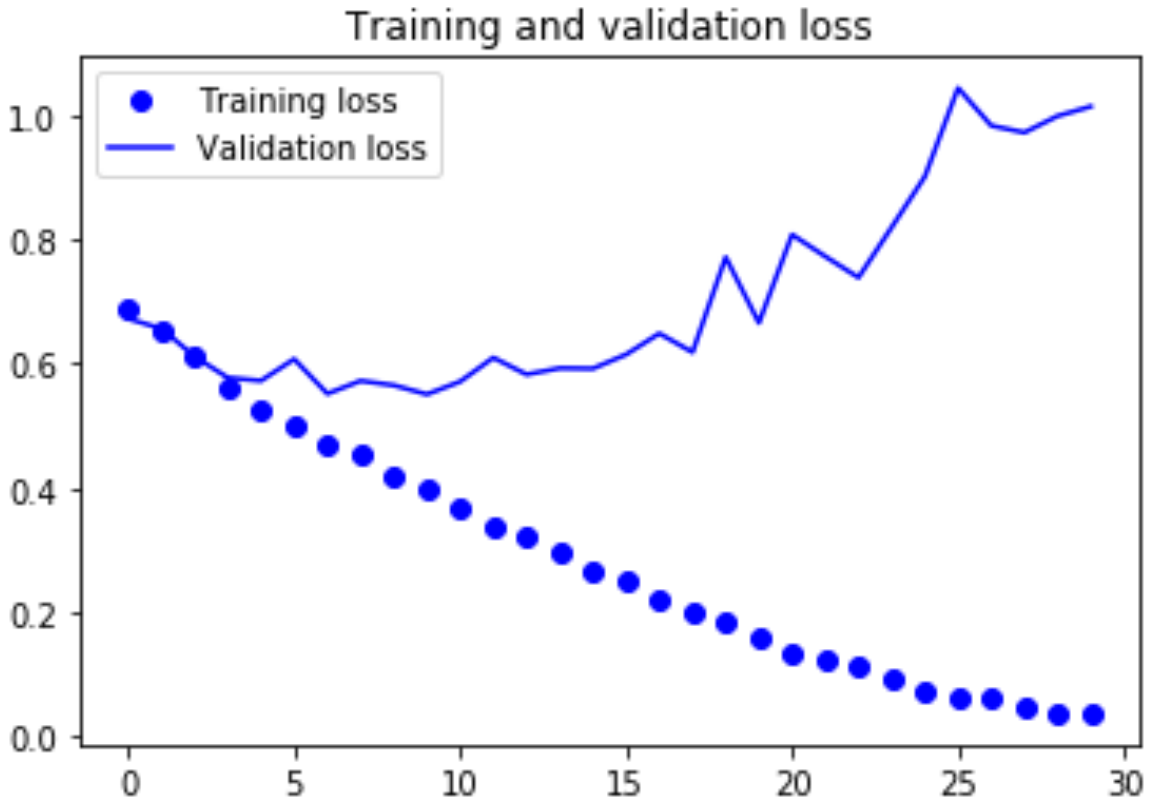
Plot the ***accuracy*** against ***epochs*** (1 epoch = 1 pass over the data).



- Training acc: 99.0%
- Validation acc: 72.4%
- Likely overfitting

4. Examine the Results

Plot the ***loss*** against ***epochs*** (1 epoch = 1 pass over the data).



- Training loss is decreasing.
- Validation loss decreases and then increase.

Why Overfitting?

```
=================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________
```

- Over $3M$ parameters; Just $2K$ training samples.
- Overfitting is not surprising.

Trick 1: Dropout

Keras's Dropout Layer

- Dropout only before the 1st dense layer to regularize the 1st dense layer.
- Because the 1st dense layer has too many trainable parameters.

```
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.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

Trick 2: Data Augmentation

Setup Data Augmentation Using 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,)
```

```
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')
```

Train the CNN

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

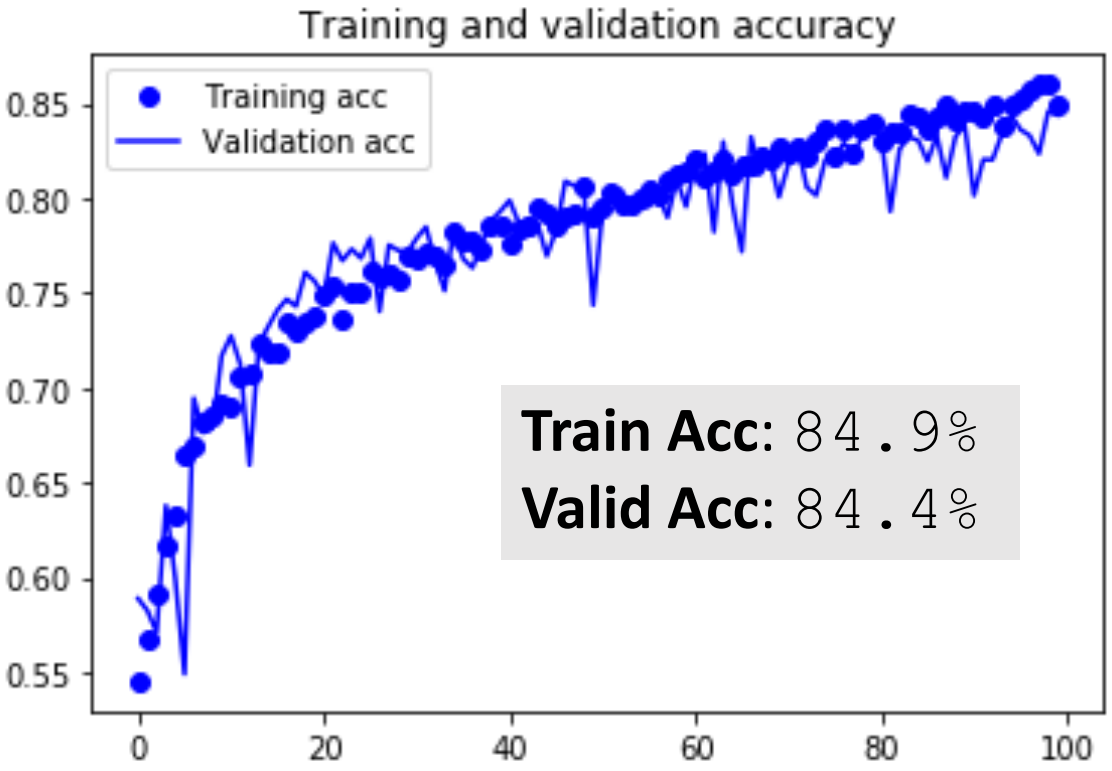
```
Epoch 1/100
100/100 [==============================] - 24s - loss: 0.6857 - acc: 0.5447 - val_loss: 0.6620 - val_acc: 0.5888
Epoch 2/100
100/100 [==============================] - 23s - loss: 0.6710 - acc: 0.5675 - val_loss: 0.6606 - val_acc: 0.5825
Epoch 3/100
100/100 [==============================] - 22s - loss: 0.6609 - acc: 0.5913 - val_loss: 0.6663 - val_acc: 0.5711
```

-
-
-

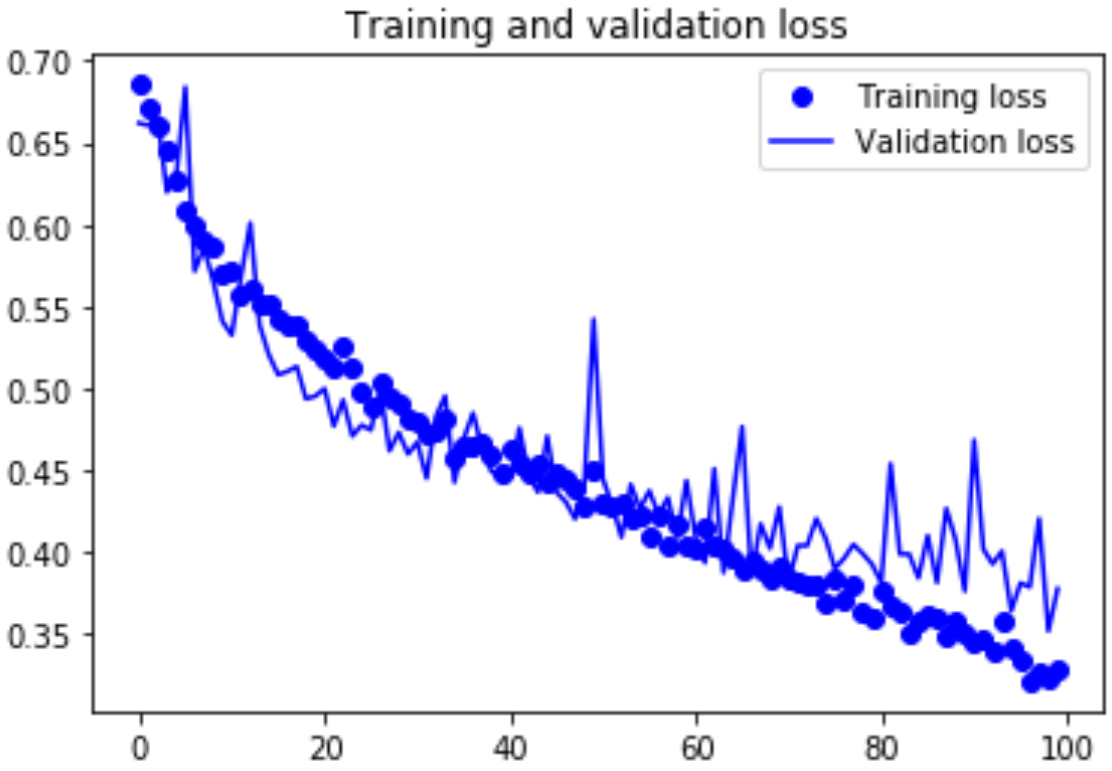
```
Epoch 99/100
100/100 [==============================] - 22s - loss: 0.3255 - acc: 0.8581 - val_loss: 0.3518 - val_acc: 0.8460
Epoch 100/100
100/100 [==============================] - 22s - loss: 0.3280 - acc: 0.8491 - val_loss: 0.3776 - val_acc: 0.8439
```

Examine the Results

accuracy against ***epochs***



loss against ***epochs***

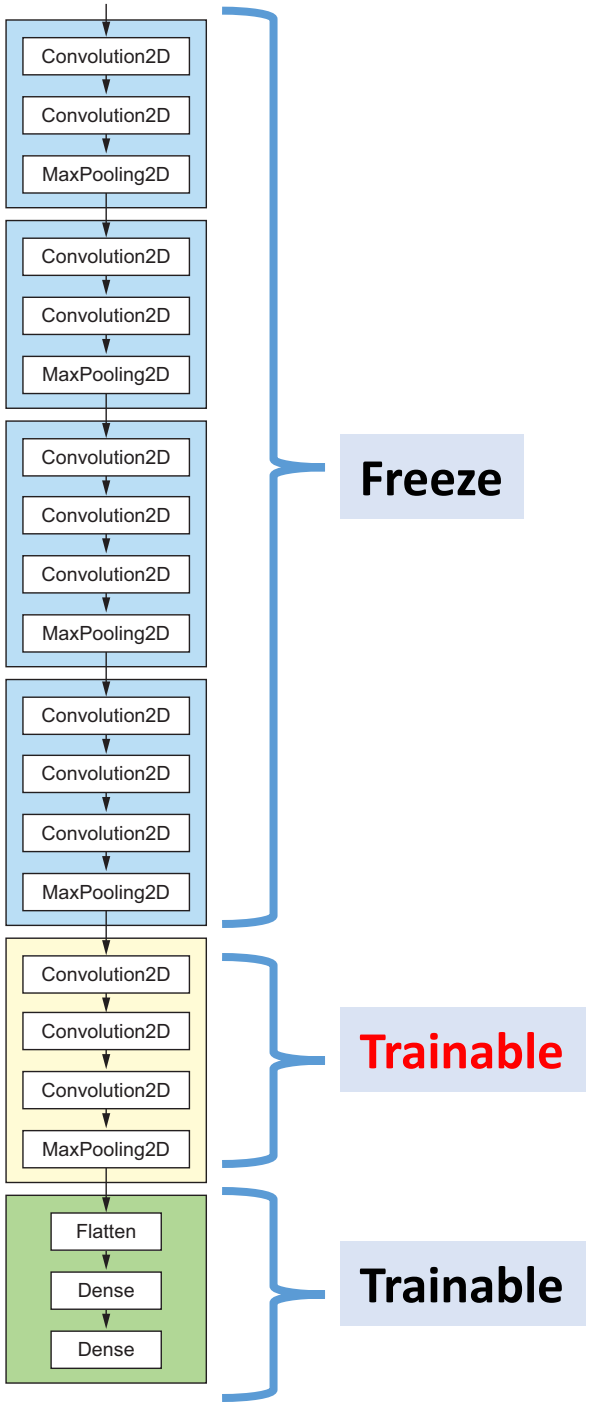


Trick 3: Pretrain

Train a Deep Neural Network?

```
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.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

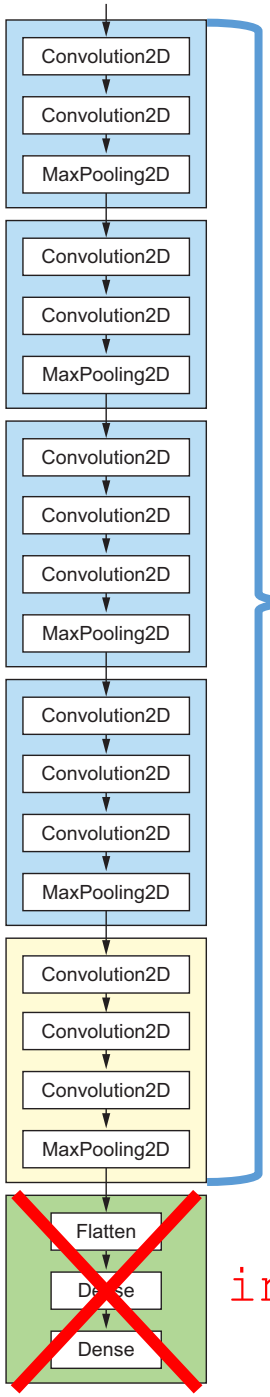
- We have trained a neural net with 4 Conv Layers and 2 FC Layers.
- Relatively shallow.



Pretrain

1. Pre-train a deep net on large-scale dataset, e.g., ImageNet ($14M$ images with labels).
2. Remove the top layers.
3. Build new top layers (randomly initialized).
4. Freeze the base layers; Train the top layers.
5. Optional: **Fine-tune** the top Conv Layers.

Use a VGG16 Net Pretrained on ImageNet

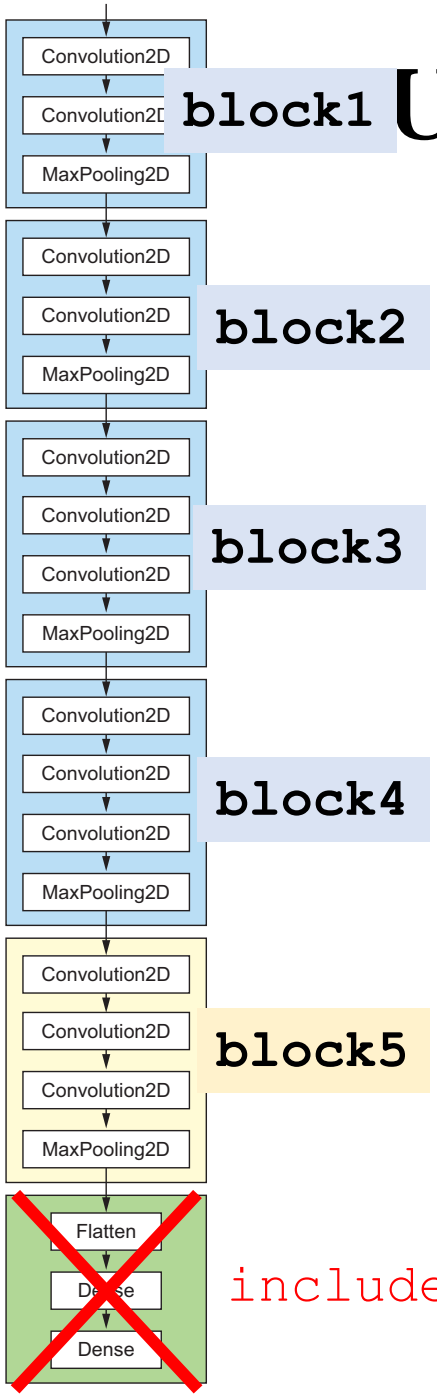


Base

```
from keras.applications import VGG16

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

conv_base.summary()
```

include_top=False

Use a VGG16 Net Pretrained on ImageNet

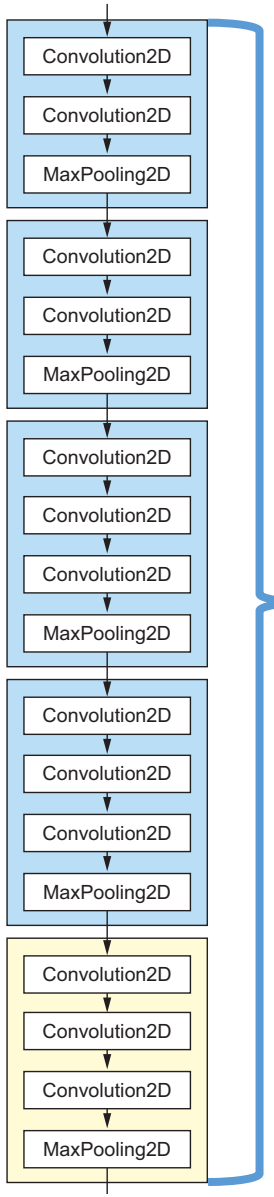


include_top=False

Layer (type)	Output Shape	Param #
input_2 (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

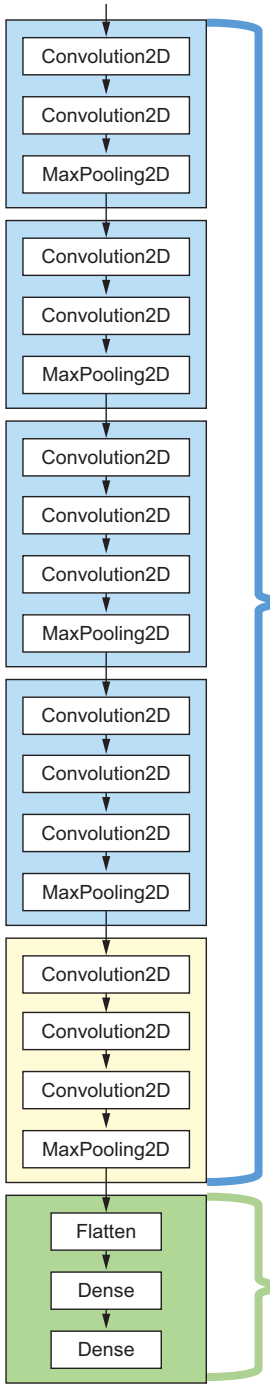
Use a VGG16 Net Pretrained on ImageNet



Base
(trainable)

```
from keras import models
from keras import layers

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

Use a VGG16 Net Pretrained on ImageNet

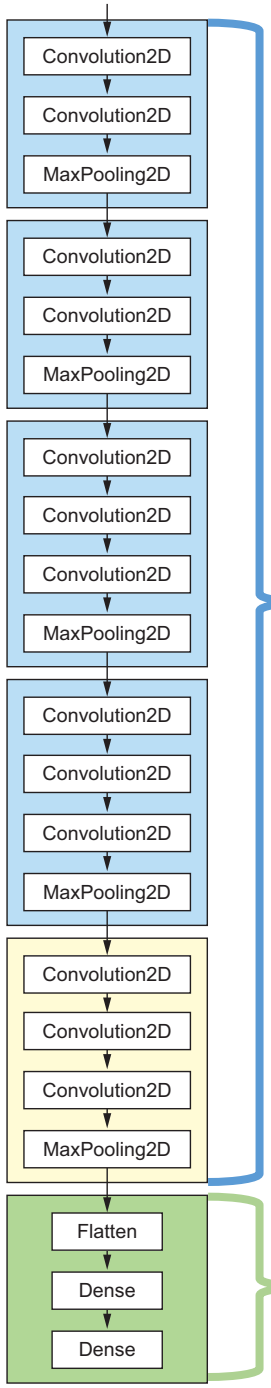


Base
(trainable)

New Top
(trainable)

```
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'))
```

Use a VGG16 Net Pretrained on ImageNet



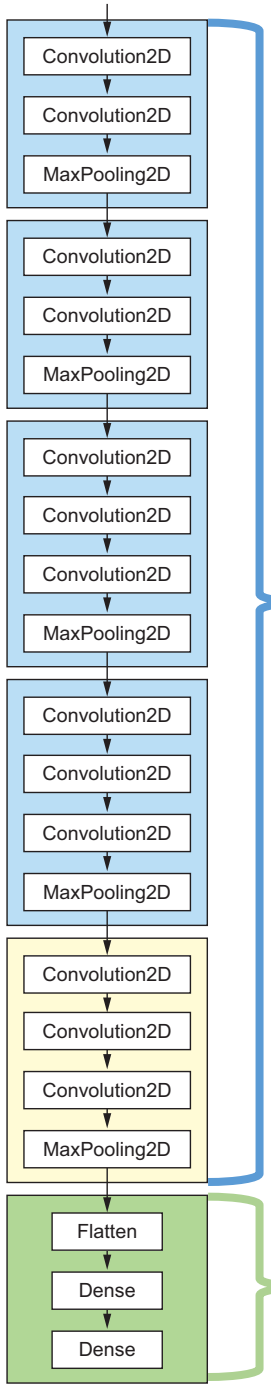
Base (trainable)

New Top (trainable)

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

Use a VGG16 Net Pretrained on ImageNet



Base (freeze)

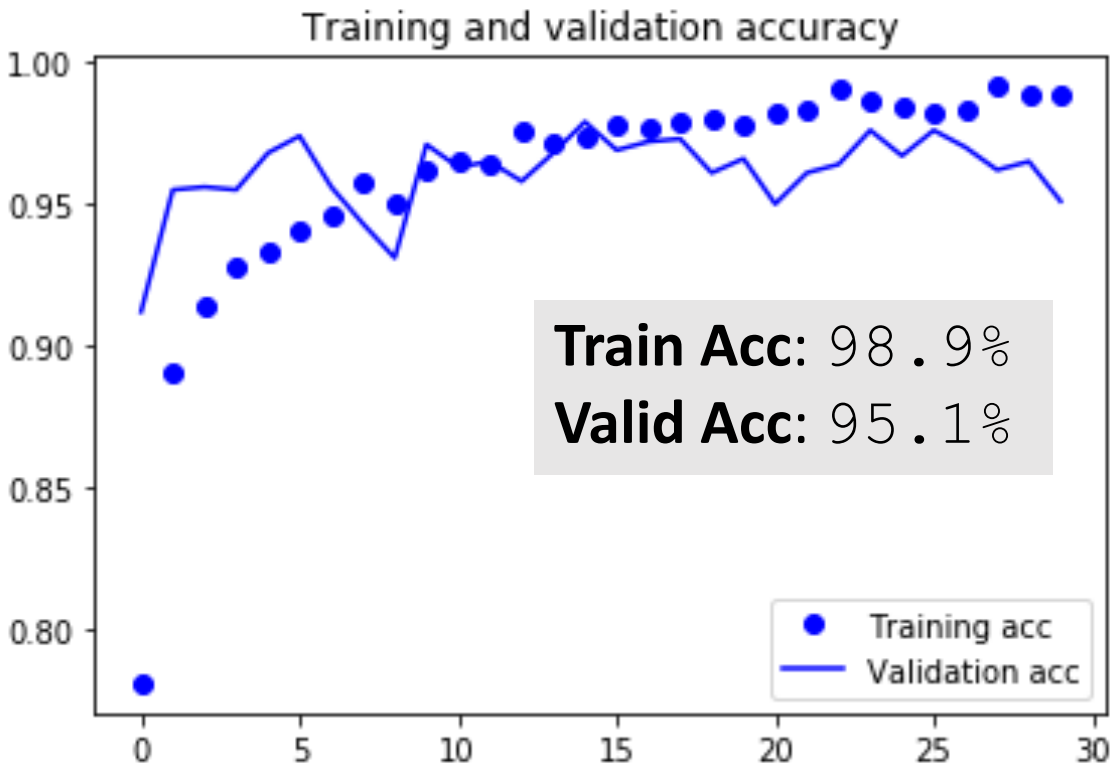
New Top (trainable)

```
conv_base.trainable = False
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: 2,097,665
Non-trainable params: 14,714,688
```

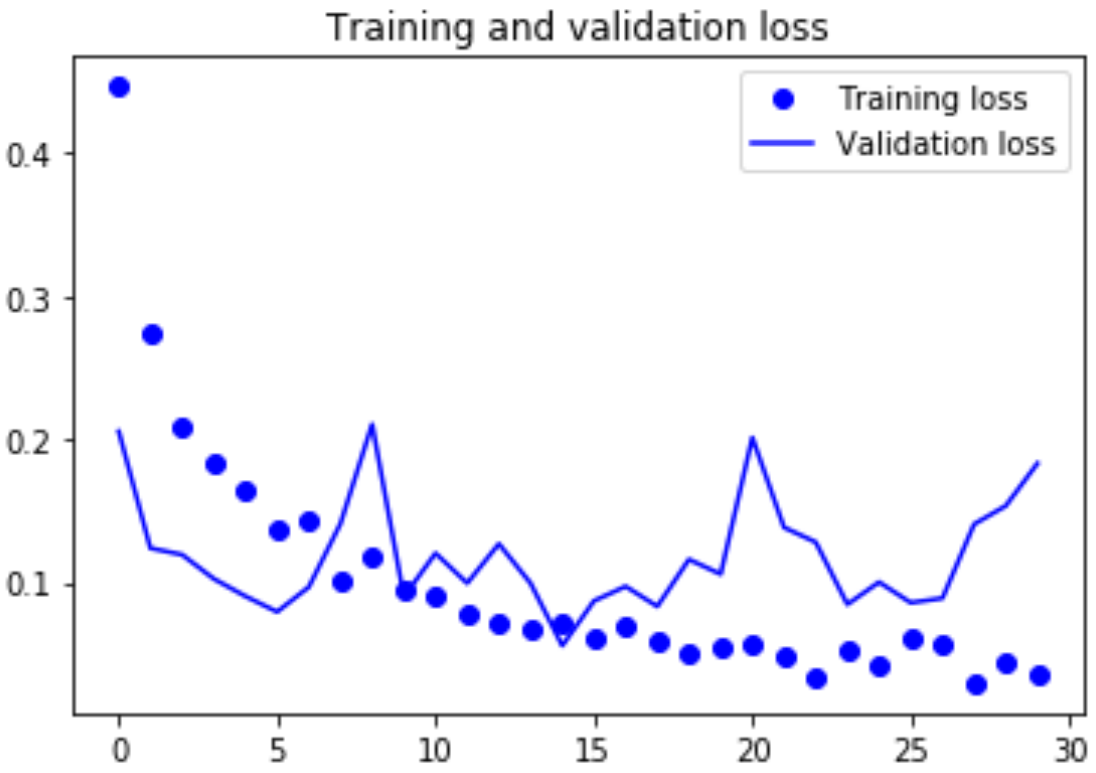
After Training the New Top

accuracy against ***epochs***

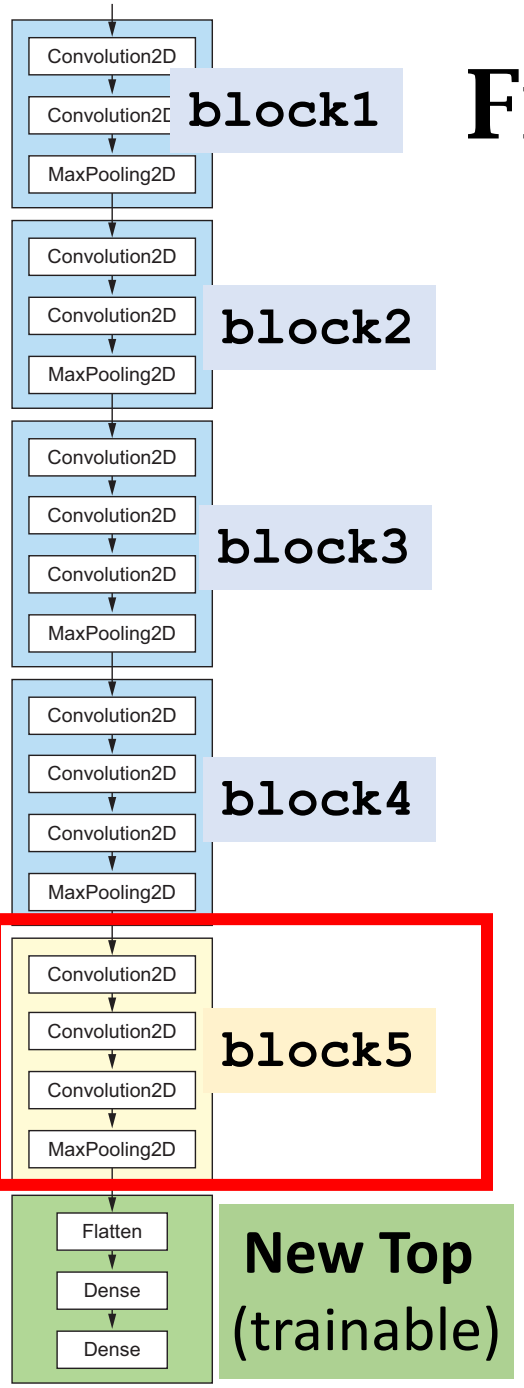


Train Acc: `98.9%`
Valid Acc: `95.1%`

loss against ***epochs***



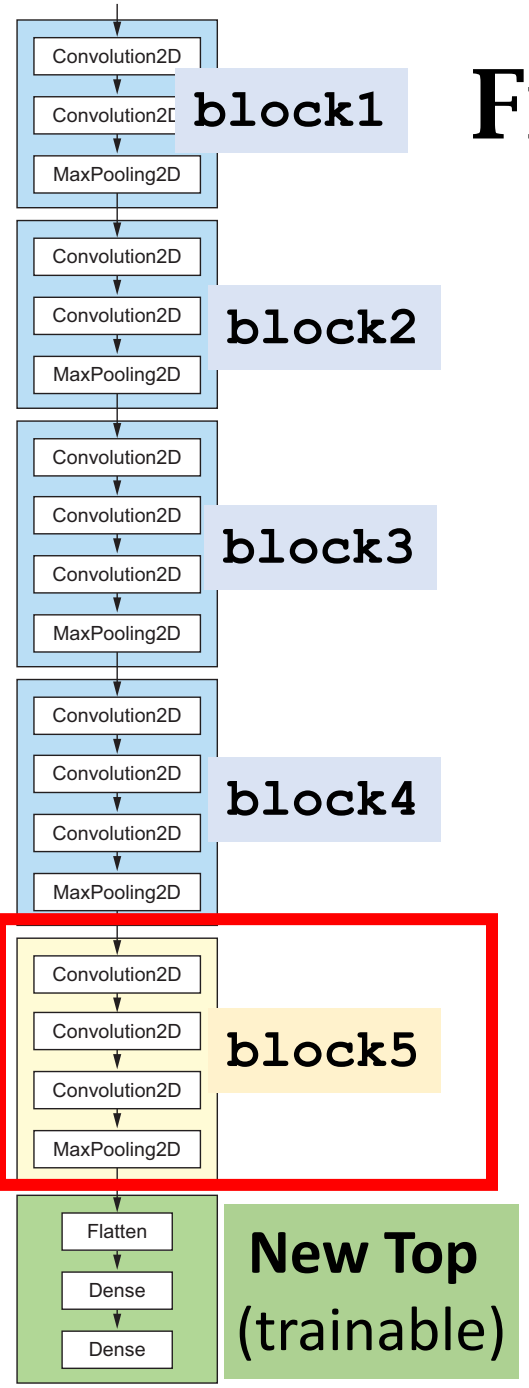
Fine Tuning the Top Conv Layers



Layer (type)	Output Shape	Param #
input_2 (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

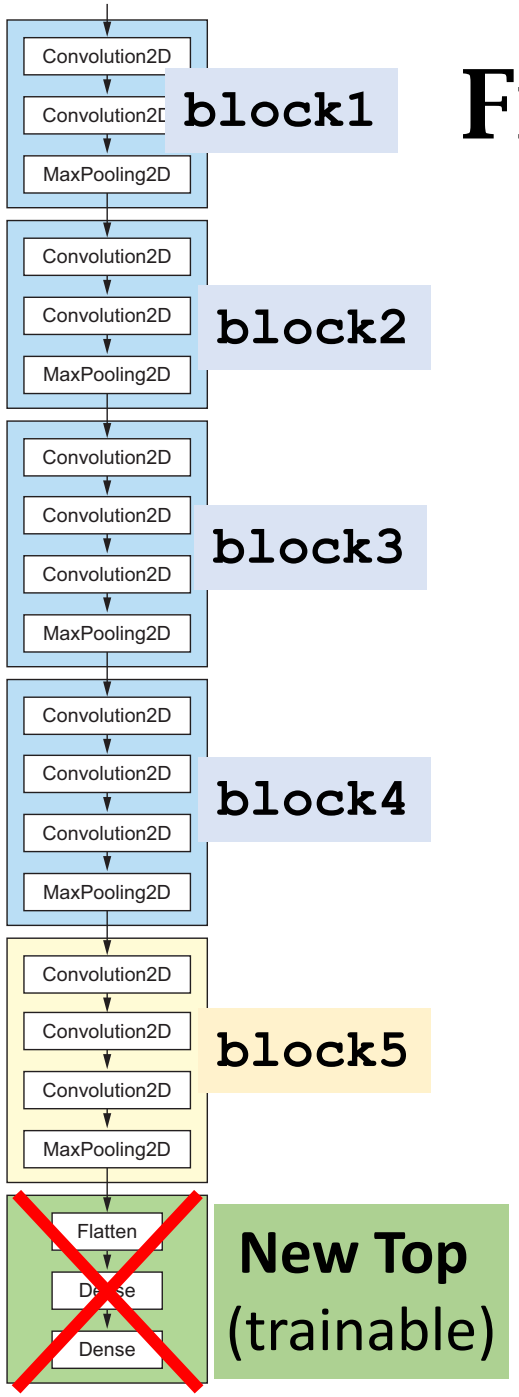
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0

Fine Tuning the Top Conv Layers




```
trainable_layer_names = ['block5_conv1', 'block5_conv2',
                         'block5_conv3', 'block5_pool']
conv_base.trainable = True

for layer in conv_base.layers:
    if layer.name in trainable_layer_names:
        layer.trainable = True
    else:
        layer.trainable = False
```

Fine Tuning the Top Conv Layers




```
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: 9,177,089
Non-trainable params: 7,635,264
_________________________________________________________________
```

Fine Tuning the Top Conv Layers

Re-compile before training

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

Fine Tuning the Top Conv Layers

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

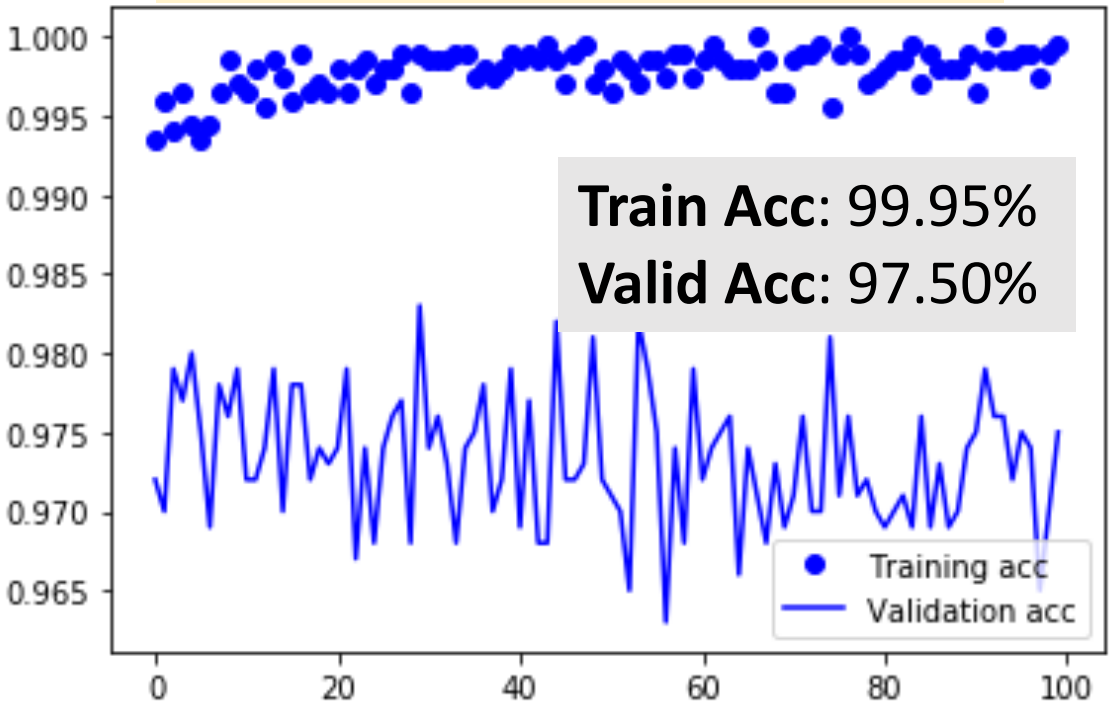
```
Epoch 1/100
100/100 [==============================] - 32s - loss: 0.0215 - acc: 0.9935 - val_loss: 0.0980 - val_acc: 0.9720
Epoch 2/100
100/100 [==============================] - 32s - loss: 0.0131 - acc: 0.9960 - val_loss: 0.1247 - val_acc: 0.9700
Epoch 3/100
100/100 [==============================] - 32s - loss: 0.0140 - acc: 0.9940 - val_loss: 0.1044 - val_acc: 0.9790
```

-
-
-

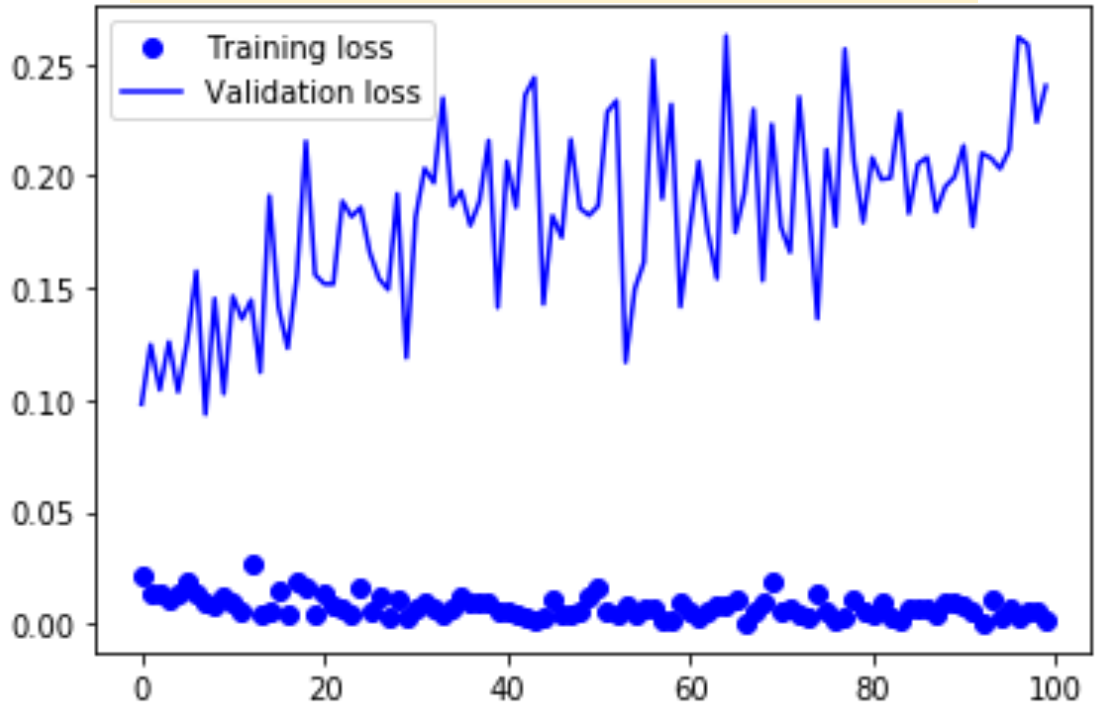
```
Epoch 99/100
100/100 [==============================] - 33s - loss: 0.0060 - acc: 0.9990 - val_loss: 0.2242 - val_acc: 0.9700
Epoch 100/100
100/100 [==============================] - 33s - loss: 0.0010 - acc: 0.9995 - val_loss: 0.2403 - val_acc: 0.9750
```

Fine Tuning the Top Conv Layers

accuracy against ***epochs***



loss against ***epochs***



Fine Tuning the Top Conv Layers

Evaluate the model on the test set

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

test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```

```
Found 1000 images belonging to 2 classes.
test acc: 0.967999992371
```

Summary of the Results

- A small ConvNet (4 Conv Layers + 2 FC Layers), with $3.5M$ parameters.
  - Training accuracy: 99.0%
  - Validation accuracy: 72.4%
- The small ConvNet + 1 Dropout Layer + Data Augmentation.
  - Training accuracy: 84.9%
  - Validation accuracy: 84.4%
- VGG16 Net pre-trained on the ImageNet. (Train the new top layers.)
  - Training accuracy: 98.9%
  - Validation accuracy: 95.1%
- VGG16 Net pre-trained on the ImageNet. (Fine-tune the top Conv Layers.)
  - Training accuracy: 99.95%
  - Validation accuracy: 97.5%