Minería de datos: clasificación de imágenes

Andrés Mañas Mañas

May 20, 2017

Contents

1 Enunciado de la práctica

En esta práctica se propone un conjunto de imágenes de polen y se pide buscar un clasificador para las mismas.

El conjunto de datos comprende 791 imágenes JPEG de 23 clases distintas de polen. Cada clase contiene aproximadamente 35 imágenes (aunque algunas de las imágenes originales han sido retiradas y por lo tanto, algunas clases pueden tener menos de 35 ejemplos).

Se trata de utilizar técnicas de deep learning para llegar a un espacio de representacion que optimice un clasificador de imágenes. El clasificador final debe ser capaz de reconocer los tipos de polen contenidos en las imágenes. Sería ideal llegar a un espacio de representacion que fuese capaz de reconocer distintos tipos de polen en una misma imagen, pero con el conjunto de entrenamiento que ponemos a vuestra disposicion, eso es imposible sin un trabajo enorme de procesado de las imagenes. Por lo tanto, basta con encontrar un clasificador que ofrezca unos resultados de clasificación decentes; entorno al 80% de acierto sería suficiente.

Es importante reflexionar sobre la necesidad de aplicar cierto tipo de preprocesado y normalización de los datos de entrada. Así mismo se debe describir las técnicas que finalmente se decidan utilizar en este ámbito.

Para la realización de la práctica se pueden utilizar muchos y variados lenguajes de programación y herramientas de desarrollo. Nosotros sugerimos aquí los siguientes:

- Python: Keras + TensorFlow
- Python: Keras + Theano
- R: R + TensorFlow
- R: R + h2o
- R: R + deepnet
- R: R + arch
- Java/Scala: Spark deeplearning4j

Se pide presentar una tabla comparando los resultados obtenidos con los distintos experimentos realizados y un apartado de conclusiones en dónde se resuman los puntos más importantes del trabajo desarrollado y las conclusiones de los mismos.

2 Preprocesado de los datos

Se han desarrollado funciones que nos permiten recortar imágenes, de modo que se obtienen imágenes cuadradas, y generar conjuntos de entrenamiento y prueba que se almacenan en disco en una estructura de directorios que permiten a Keras interpretar a cada imagen en el tipo de conjunto al que pertenece (entrenamiento/prueba) y en la categoría o clase con la que se corresponde.

El código de esta parte de la práctica puede verse aquí.

En este sentido he puesto especial atención a la necesidad de que las imágenes tienen que tener todas la misma dimensión (he pensado que cuadradas). Por eso he implementado unas pequeñas funciones que nos permiten, dada una imagen, tomar los dos extremos cuadrados más grandes de la misma. por ejemplo, si tenemos una imagen de 2x3, nos quedaríamos con el cuadrado 2x2 más arriba y a la izquierda y el cuadrado 2x2 más abajo y a la derecha.

Hay que tener en cuenta que este método hace que una imagen que ya de por sí sea cuadrada o casi cuadrada acaba apareciendo "duplicada" en el conjunto de entrada.

Siguiendo estas técnicas, el conjunto de datos original se ha transformado en éste

Aparte de esto, dado que el conjunto de imágenes es muy pequeño, se han utilizado técnicas de aumento de datos de entrenamiento (aplicar rotaciones, simetrías, etc.) para conseguir mayor diversidad de datos y evitar el sobreentrenamiento.

3 Clasificación con un perceptrón multicapa sencillo

Esta sección la incluyo únicamente para tener un experimento sobre el que comparar si el resto del trabajo realizado mejora o no un clasificador tomado sin mayor esfuerzo de selección o de afinado.

Como la práctica la realizo con Keras y Tensorflow, básicamente lo que hago aquí es coger el clasificador más sencillo que encuentro en la documentación.

3.1 Diseñamos y compilamos el modelo

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense

train_data_dir = '../data/train'
validation_data_dir = '../data/validation'

width, height = 200, 200
train_samples = 1152
validation_samples = 288
categories = 21
batch_size = 4
epochs = 20


def modelo_de_contraste():
    model = Sequential()
    model.add(Convolution2D(32, 3, 3, input_shape=(width, height, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Convolution2D(32, 3, 3))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Convolution2D(64, 3, 3))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(64))
    model.add(Activation('relu'))
    model.add(Dropout(0.5))
```

```
    model.add(Dense(categories, activation='softmax'))

    model.compile(
        loss='categorical_crossentropy',
        optimizer='rmsprop',
        metrics=['accuracy'])
    return model
```

3.2 Lo entrenamos

```
import os
from keras.models import load_model

train_datagen = ImageDataGenerator(rescale=1. / 255)
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    batch_size=batch_size,
    class_mode='categorical',
    target_size=(width, height),
    shuffle=False)

validation_datagen = ImageDataGenerator(rescale=1. / 255)
validation_generator = validation_datagen.flow_from_directory(
    validation_data_dir,
    batch_size=batch_size,
    class_mode='categorical',
    target_size=(width, height),
    shuffle=False)

if not os.path.isfile('modelo_de_contraste.h5'):
    model = modelo_de_contraste()
    model.fit_generator(
        train_generator,
        samples_per_epoch=128,
        nb_epoch=epochs,
        validation_data=validation_generator,
        nb_val_samples=32,
        verbose=2)
    model.save('modelo_de_contraste.h5')
else:
    model = load_model('modelo_de_contraste.h5')
```

```
Found 1152 images belonging to 21 classes.
Found 288 images belonging to 21 classes.
```

3.3 Y evaluamos el resultado en el conjunto de prueba

```
score = model.evaluate_generator(validation_generator, validation_samples)

print('Model loss (categorical_crossentropy):', score[0])
print('Model accuracy:', score[1])
```

```
('Model loss (categorical_crossentropy):', 12.055745086671473)
('Model accuracy:', 0.048611111111111112)
```

Vemos que el resultado es realmente decepcionante.

Cualquier cosa que hagamos seguramente mejorará estos resultados.

4 Fine tunning de redes pre-entranadas

Mi propuesta para acometer la labor de clasificar las imágenes de polen es utilizar redes ya entrenadas y servirnos de ellas para con pocas modificaciones utilizarlas en nuestras imágenes.

Los experimentos realizados en este notebook se basan en las indicaciones de este blog

La idea, básimente consiste en:

1. Coger una red ya entrenada previamente y quitarle la capa superior
2. Clasificar nuestro conjunto de datos con la red resultante del paso anterior
3. Diseñar un modelo sencillo cuyo input es el output del punto 2 y entrenarlo

Aparentemente con muy poco cálculo se pueden obtener buenos resultados.

En los siguientes experimentos voy a probar el planteamiento anterior utilizando las redes pre-entrenadas que vienen pon defecto con Keras para ver cual de ellas ofrece mejores resultados.

Después, una vez seleccionada una, intentaré determinar y optimizar el diseño del modelo superior.

4.1 Parámetros comunes para todos los experimentos

```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
import os
import os.path
import json
import requests
from os.path import expanduser
import tarfile
from keras import applications
from keras.layers import Dropout, Flatten, Dense, Input
from keras.models import Sequential, load_model
from keras.preprocessing.image import ImageDataGenerator


data_dir = '../data/{}'

intermediate_dropbox = 'https://www.dropbox.com/s/fsqh2emxq8kp80i/intermediate-data.tar.gz?dl=1'
top_model_dropbox = 'https://www.dropbox.com/s/bwxbjzkpi2drjmi/top_model_results.tar.gz?dl=1'
```

```
cache_dir = expanduser("~") + '/DMdeeplearning'
intermediate_tar = cache_dir + '/intermediate-data.tar.gz'
top_model_tar = cache_dir + '/top_model_results.tar.gz'
intermediate_file_path = cache_dir + '/{}_{}_{}.npy'
top_model_path = cache_dir + '/{}_top_model.h5'
history_path = cache_dir + '/{}_history.json'
```

4.2 Caché de datos intermedios

Los datos intermedios sólo es necesario generarlos una vez por modelo CNN.

Para no andar recalculándolos cada vez que se pruebe un top model, guardo los resultados de mis pruebas en dropbox e implemento funcionalidades que los descargan y utilizan automáticamente si ya existen.

```
def download_dropbox_data():
    if not os.path.exists(cache_dir):
        os.makedirs(cache_dir)

    if not os.path.exists(intermediate_tar):
        print 'downloading cached data (will take a while) ... '
        r = requests.get(intermediate_dropbox, stream=True)
        with open(intermediate_tar, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1024):
                if chunk:
                    f.write(chunk)
        tarfile.open(intermediate_tar).extractall(cache_dir)

    if not os.path.exists(top_model_tar):
        print 'downloading cached data (will take a while) ... '
        r = requests.get(top_model_dropbox, stream=True)
        with open(top_model_tar, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1024):
                if chunk:
                    f.write(chunk)
        tarfile.open(top_model_tar).extractall(cache_dir)

def already_cached(file_path):
    return os.path.isfile(file_path)

def get_from_cache(file_path):
    if already_cached(file_path):
        return np.load(file_path)

download_dropbox_data()
```

4.3 Generación de datos intermedios

Definimos unas funciones que, dado un modelo pre-entrenado, permiten traducir nuestros datos en carácterísticas y etiquetas para utilizarse en el top model.

```
def generate_intermediate_data(model, num_samples, data_type):
    """ data_type puede ser train o validation (alguno de los
    subdirectorios de datos)
    """

    name = model.name

    features_path = intermediate_file_path.format(name, data_type, 'features')
    labels_path = intermediate_file_path.format(name, data_type, 'labels')

    if (already_cached(features_path) and already_cached(labels_path)):
        print 'intermediate %s data for %s already generated' \
            % (data_type,name)
        return

    print 'predicting features and labels for %s' % name

    naive_datagen = ImageDataGenerator(rescale=1. / 255)
    dataflow = naive_datagen.flow_from_directory(
        data_dir.format(data_type),
        batch_size=batch_size,
        class_mode='categorical',
        target_size=(width, height),
        shuffle=False)

    features = None
    labels = None
    rounds = num_samples // batch_size
    print 'running {} rounds'.format(rounds)
    for i in range(rounds):
        if i % 50 == 0:
            print
            print i, '/', rounds, '.',
        else:
            print '.',
        batch = dataflow.next()
        batch_features = model.predict(batch[0])
        batch_labels = batch[1]

        if features is None:
            features = batch_features
        else:
            features = np.append(features, batch_features, axis=0)

        if labels is None:
            labels = batch_labels
        else:
            labels = np.append(labels, batch_labels, axis=0)
    print 'done'

    np.save(open(features_path, 'w'), features)
    np.save(open(labels_path, 'w'), labels)
```

4.4 Renderizado

Definimos unas utilidades para dibujar métricas de los experimentos.

```
def plot_history(name):

    history = json.load(open(history_path.format(name)))

    plt.figure(figsize=(12, 4))

    # summarize history for accuracy
    plt.subplot(121)
    plt.plot(history['acc'])
    plt.plot(history['val_acc'])
    plt.title('Figure: ' + name + ' model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.grid(True)

    # summarize history for loss
    plt.subplot(122)
    plt.plot(history['loss'])
    plt.plot(history['val_loss'])
    plt.title('Figure: ' + name + ' model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.grid(True)

    plt.suptitle('Figure: metrics for ' + name,y=1.05,fontsize=16)
    plt.tight_layout()
    plt.show()


def plot_histories(names):

    plt.figure(figsize=(24, 8))

    for i in range(len(names)):
        name = names[i]
        history = json.load(open(history_path.format(name)))
        plt.subplot(121)
        plt.plot(history['acc'])
        plt.plot(history['val_acc'])

    plt.title('Figure: model accuracies',fontsize=16)
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend([j for i in [[name + ' train', name + ' test']
                            for name in names]
                for j in i],
               loc='upper left')
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    plt.figure(figsize=(24, 8))

    for i in range(len(names)):
        name = names[i]
        history = json.load(open(history_path.format(name)))
        plt.subplot(121)
        plt.plot(history['loss'])
        plt.plot(history['val_loss'])

    plt.title('Figure: model losses',fontsize=16)
```

```
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend([j for i in [[name + ' train', name + ' test']
                            for name in names]
                for j in i],
               loc='upper left')
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

4.5 Lanzador de experimentos

4.5.1 Top model común

Una capa densa de 256 nodos seguida de un softmax sobre las categorías de imágenes que tenemos.

Con este modelo superior común a todas las redes preentrenadas descabezadas, veremos cuál ofrece mejor rendimiento.

```
def common_top_model(input_shape):
    model = Sequential()
    model.add(Flatten(input_shape=input_shape))
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(categories, activation='softmax'))
    return model
```

4.5.2 Lanzador

Dado un modelo preentrenado y un top model, y asumiendo que nuestros datos de entrenamiento y validación han sido clasificados con el modelo preentrenado, entrena el top model.

Guarda en disco el top model entrenado y el historial de evolución de rendimiento, para posteriormente poderlo representar.

No hace nada si previamente ya se ha realizado un experimento que tenga el mismo nombre que el que se está intentando realizar.

```
def run_experiment(exp_name, pretrained_model, top_model=None, epochs=epochs):

    print 'starting %s experiment' % exp_name

    # stops if already done
    if os.path.isfile(top_model_path.format(exp_name)):
        print 'experiment %s is already done' % exp_name
        return

    pmn = pretrained_model.name
    tfp = intermediate_file_path.format(pmn, 'train', 'features')
```

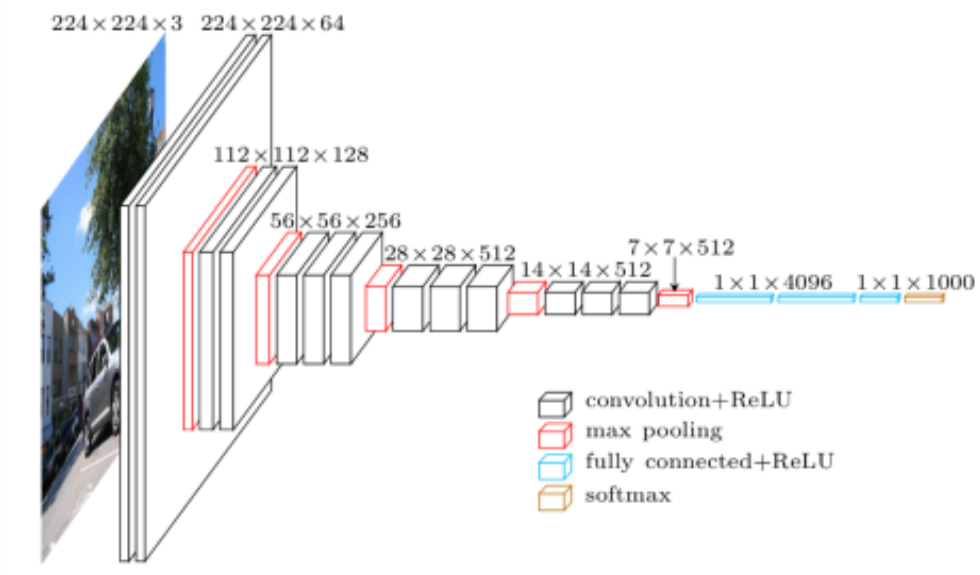
```
    tlp = intermediate_file_path.format(pmn, 'train', 'labels')
    vfp = intermediate_file_path.format(pmn, 'validation', 'features')
    vlp = intermediate_file_path.format(pmn, 'validation', 'labels')

    print 'loading intermediate data for %s' % pretrained_model.name
    train_features = get_from_cache(tfp)
    train_labels = get_from_cache(tlp)
    validation_features = get_from_cache(vfp)
    validation_labels = get_from_cache(vlp)

    print 'shapes for %s: ' % exp_name
    print '\t', train_features.shape
    print '\t', train_labels.shape
    print '\t', validation_features.shape
    print '\t', validation_labels.shape

    print 'training top model for %s' % exp_name

    if top_model is None:
        top_model = common_top_model(train_features.shape[1:])
        top_model.compile(
            optimizer='rmsprop',
            loss='categorical_crossentropy',
            metrics=['accuracy'])

    history = top_model.fit(train_features,
                            train_labels,
                            batch_size=batch_size,
                            nb_epoch=epochs,
                            validation_data=(validation_features,
                                             validation_labels))

    print 'saving top model for %s' % exp_name
    top_model.save(top_model_path.format(exp_name))

    print 'saving history for %s' % exp_name
    json.dump(history.history, open(history_path.format(exp_name), 'w'))

    print '%s done!' % exp_name

def evaluate_top_model(top_model_file,pretrained_model_name):
    pmn = pretrained_model_name
    vfp = intermediate_file_path.format(pmn, 'validation', 'features')
    vlp = intermediate_file_path.format(pmn, 'validation', 'labels')
    validation_features = get_from_cache(vfp)
    validation_labels = get_from_cache(vlp)
    model = load_model(top_model_path.format(top_model_file))
    return model.evaluate(validation_features, validation_labels,verbose=0)
```

4.6 Experimentos "naive" con distintas redes pre-entrenadas

4.6.1 VGG16

La arquitectura de red VGG fue introducida por Simonyan y Zisserman en su artículo de 2014, Very Deep Convolutional Networks para el reconocimiento de imágenes a gran escala.

Esta red se caracteriza por su simplicidad, utilizando sólo 3×3 capas convolucionales apiladas una encima de la otra en profundidad creciente. La reducción del tamaño del volumen se maneja mediante MaxPooling. Dos capas completamente conectadas, cada una con 4.096 nodos, son seguidas por un clasificador softmax.



VGG

Figura: arquitecturas VGG

Los números 16 o 19 se refieren al número de capas antes de las capas totalmente conectadas finales.

```
input_tensor=Input(shape=(width,height,3))
vgg16 = applications.VGG16(include_top=False, weights='imagenet',
                           input_tensor=input_tensor)
vgg16.name = 'VGG16'
generate_intermediate_data(vgg16,train_samples,'train')
generate_intermediate_data(vgg16,validation_samples,'validation')

run_experiment('VGG16_naive',vgg16)
loss,acc = evaluate_top_model('VGG16_naive','VGG16')
print 'VGG16 loss in test data:     %s' % loss
print 'VGG16 accuracy in test data: %s' % acc
```

```
intermediate train data for VGG16 already generated
intermediate validation data for VGG16 already generated
starting VGG16_naive experiment
experiment VGG16_naive is already done
VGG16 loss in test data:     1.16762843314
VGG16 accuracy in test data: 0.763888888889
```

4.6.2 VGG19

```
input_tensor=Input(shape=(width,height,3))
model = applications.VGG19(include_top=False, weights='imagenet',
                           input_tensor=input_tensor)
model.name = 'VGG19'
generate_intermediate_data(model,train_samples,'train')
generate_intermediate_data(model,validation_samples,'validation')

run_experiment('VGG19_naive',model)
loss,acc = evaluate_top_model('VGG19_naive','VGG19')
print 'VGG19 loss in test data:     %s' % loss
print 'VGG19 accuracy in test data: %s' % acc
```

```
intermediate train data for VGG19 already generated
intermediate validation data for VGG19 already generated
starting VGG19_naive experiment
experiment VGG19_naive is already done
VGG19 loss in test data:     0.953504317337
VGG19 accuracy in test data: 0.75
```

4.6.3 InceptionV3

Inception-v3 es la red desarrollada por Google y entrenada para el concurso ImageNet utilizando los datos de 2012. Se dice que este modelo tiene un alto rendimiento de clasificación. El objetivo perseguido por los ingenieros de Google en el desarrollo de este modelo era reducir el número de parámetros necesarios en la red para hacerle disponible en dispositovos móviles.

La red se compone de módulos llamados **inception** cuyo objetivo es actuar como un "extractor de características multinivel" al calcular convoluciones 1×1, 3×3 y 5×5 dentro del mismo módulo de la red. La salida de estos filtros se apilan a lo largo del canal de dimensión y antes de ser introducido en la siguiente capa de la red.

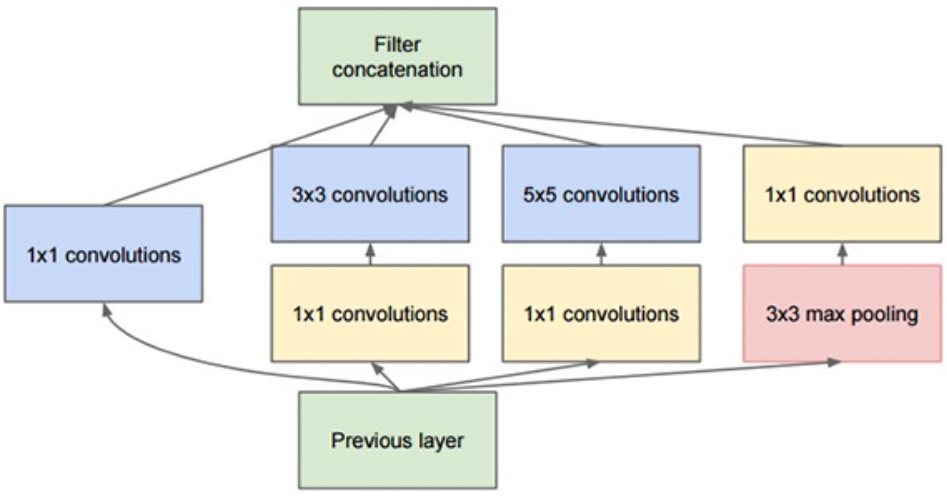
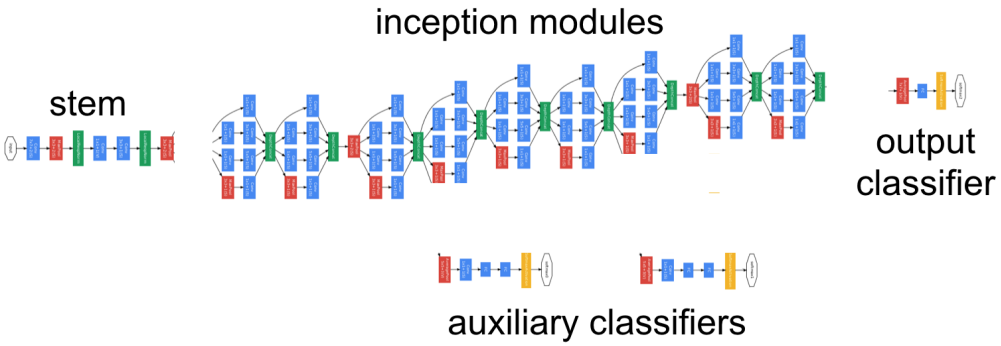


Figure: módulo inceptión

El modelo Inception-v3 pre-entrenado alcanza la precisión más avanzada para reconocer objetos generales con 1000 clases, como "Zebra", "Dalmatian" y "Dishwasher". El modelo extrae

características generales de las imágenes de entrada en la primera parte y las clasifica basándose en esas características en la segunda parte.



InceptionV3 net architecture

Figure: arquitectura de la red completa

```
input_tensor=Input(shape=(width,height,3))
model = applications.InceptionV3(include_top=False, weights='imagenet',
                                 input_tensor=input_tensor)
model.name = 'InceptionV3'
generate_intermediate_data(model,train_samples,'train')
generate_intermediate_data(model,validation_samples,'validation')

run_experiment('InceptionV3_naive',model)
loss,acc = evaluate_top_model('InceptionV3_naive','InceptionV3')
print 'InceptionV3 loss in test data:     %s' % loss
print 'InceptionV3 accuracy in test data: %s' % acc
```

```
intermediate train data for InceptionV3 already generated
intermediate validation data for InceptionV3 already generated
starting InceptionV3_naive experiment
experiment InceptionV3_naive is already done
InceptionV3 loss in test data:     5.69343353642
InceptionV3 accuracy in test data: 0.638888888889
```

4.6.4 ResNet50

A diferencia de las arquitecturas de redes secuenciales tradicionales como AlexNet, OverFeat y VGG, ResNet es en cambio una forma de "arquitectura exótica" que se basa en módulos de micro-arquitectura (también denominados "arquitecturas de red en red").

El término micro-arquitectura se refiere al conjunto de "bloques de construcción" utilizados para construir la red. La colección de bloques de construcción de micro-arquitectura (convoluciones, poolings, etc.) conduce a la macro-arquitectura.

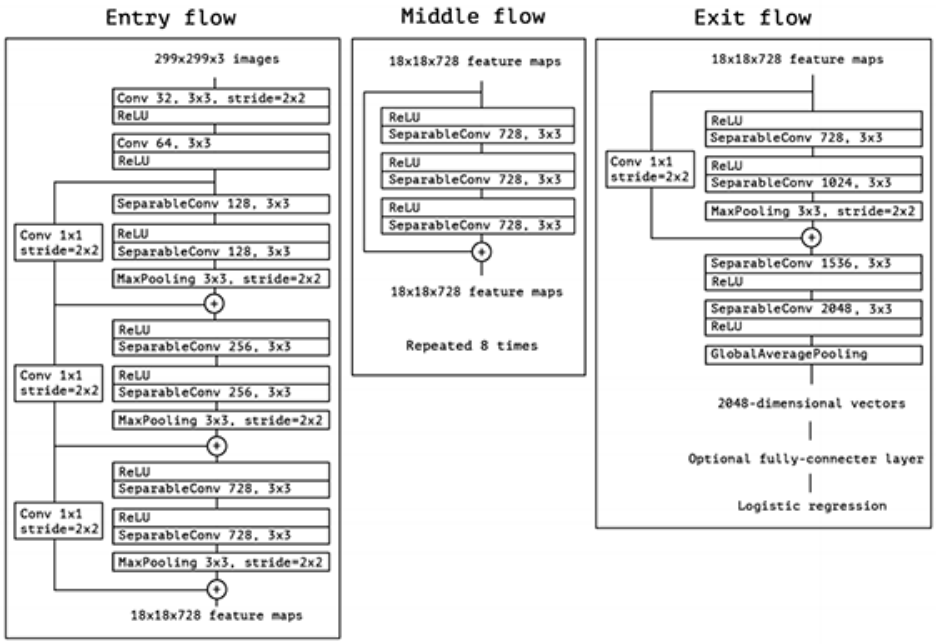
```
input_tensor=Input(shape=(width,height,3))
model = applications.ResNet50(include_top=False, weights='imagenet',
                              input_tensor=input_tensor)
model.name = 'ResNet50'
generate_intermediate_data(model,train_samples,'train')
generate_intermediate_data(model,validation_samples,'validation')

run_experiment('ResNet50_naive',model)
loss,acc = evaluate_top_model('ResNet50_naive','ResNet50')
print 'ResNet50 loss in test data:     %s' % loss
print 'ResNet50 accuracy in test data: %s' % acc
```

```
intermediate train data for ResNet50 already generated
intermediate validation data for ResNet50 already generated
starting ResNet50_naive experiment
experiment ResNet50_naive is already done
ResNet50 loss in test data:     3.03949022293
ResNet50 accuracy in test data: 0.0486111111111
```

4.6.5 Xception

Xception fue propuesta por François Chollet, el creador y jefe de mantenimiento de la biblioteca de Keras.

Xception es una extensión de la arquitectura Inception que reemplaza los módulos de inicio estándar con convoluciones separables en profundidad.



Exception

Figure: arquitectura Xception

```
input_tensor=Input(shape=(width,height,3))
model = applications.Xception(include_top=False, weights='imagenet',
                              input_tensor=input_tensor)
model.name = 'Xception'
generate_intermediate_data(model,train_samples,'train')
generate_intermediate_data(model,validation_samples,'validation')

run_experiment('Xception_naive',model)
run_experiment('Xception_naive',model)
loss,acc = evaluate_top_model('Xception_naive','Xception')
print 'Xception loss in test data:     %s' % loss
print 'Xception accuracy in test data: %s' % acc
```

```
intermediate train data for Xception already generated
intermediate validation data for Xception already generated
starting Xception_naive experiment
experiment Xception_naive is already done
starting Xception_naive experiment
experiment Xception_naive is already done
Xception loss in test data:     6.81795869933
Xception accuracy in test data: 0.5625
```

La generación de los datos temporales para cada modelo de los anteriores sólo es necesario realiarla un ver para un mismo datagen. Lleva bastante tiempo y genera archivos grandes, que pueden saturar el repositorio en github.

Por eso, los he generado yo una sóla vez y están disponibles en dropbox: intermediate-data y top_model_results.

4.7 Resultado de los experimentos "naive"

```
plot_history('VGG16_naive')
print
plot_history('VGG19_naive')
print
plot_history('InceptionV3_naive')
print
plot_history('ResNet50_naive')
print
plot_history('Xception_naive')
```

Figure: metrics for VGG16_naive

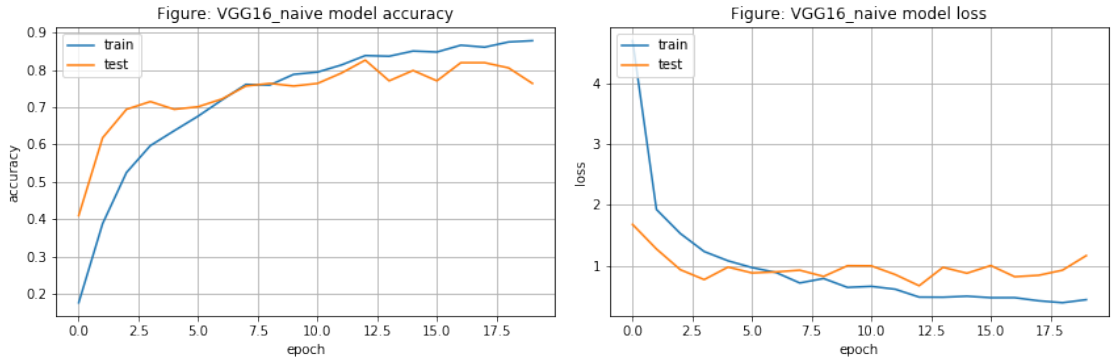


Figure: metrics for VGG19_naive

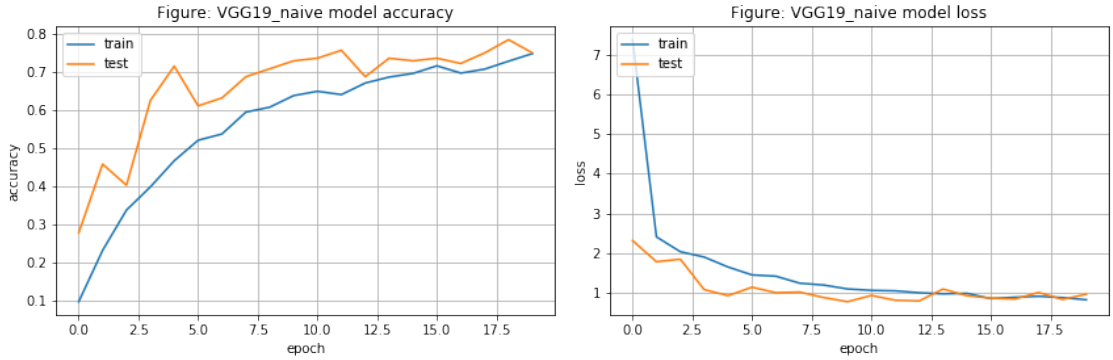


Figure: metrics for InceptionV3_naive

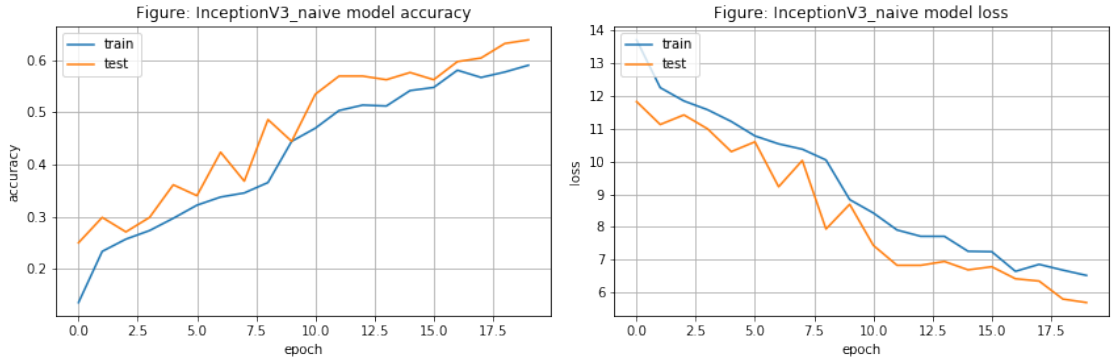


Figure: metrics for ResNet50_naive

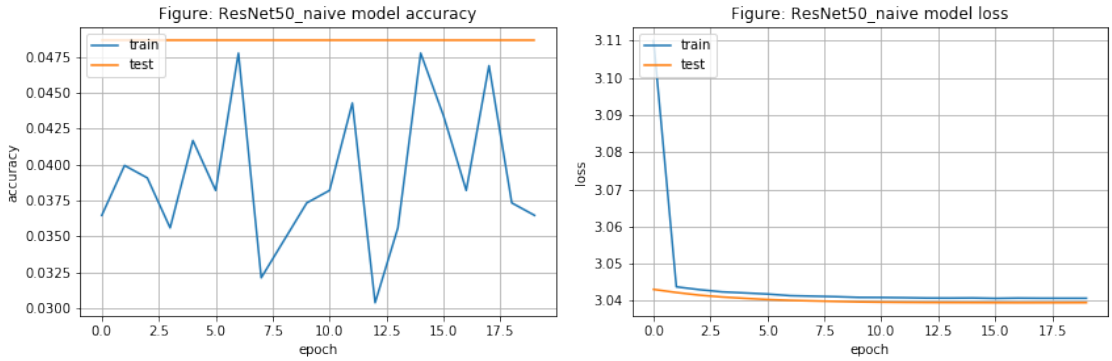
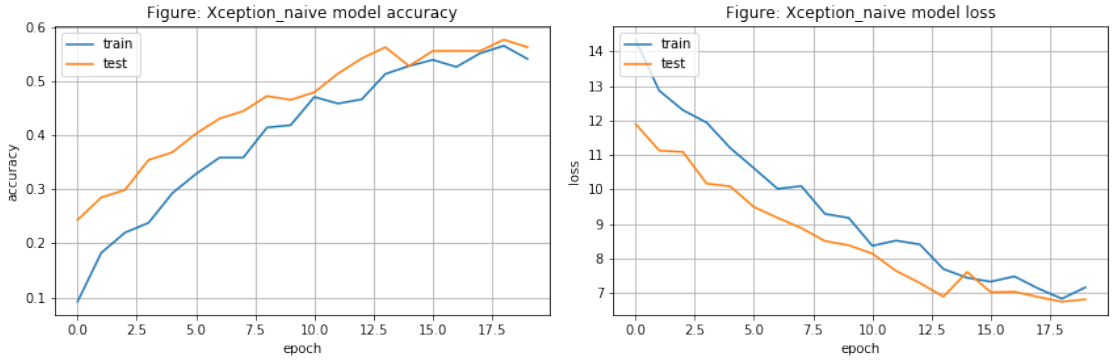


Figure: metrics for Xception_naive

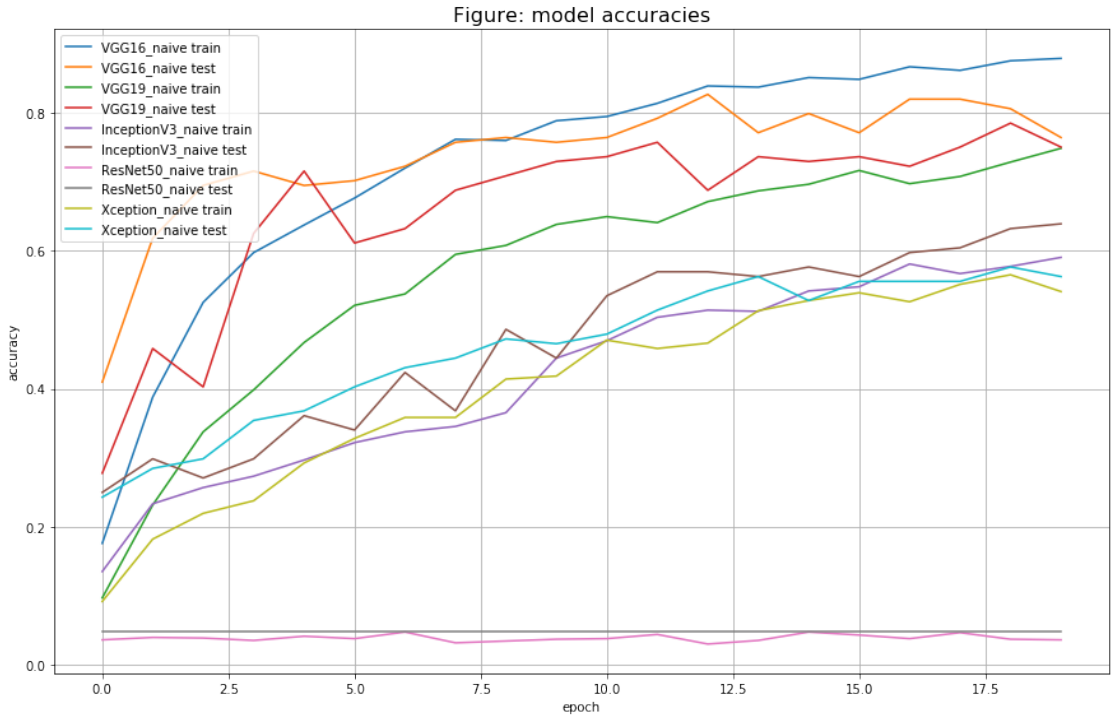


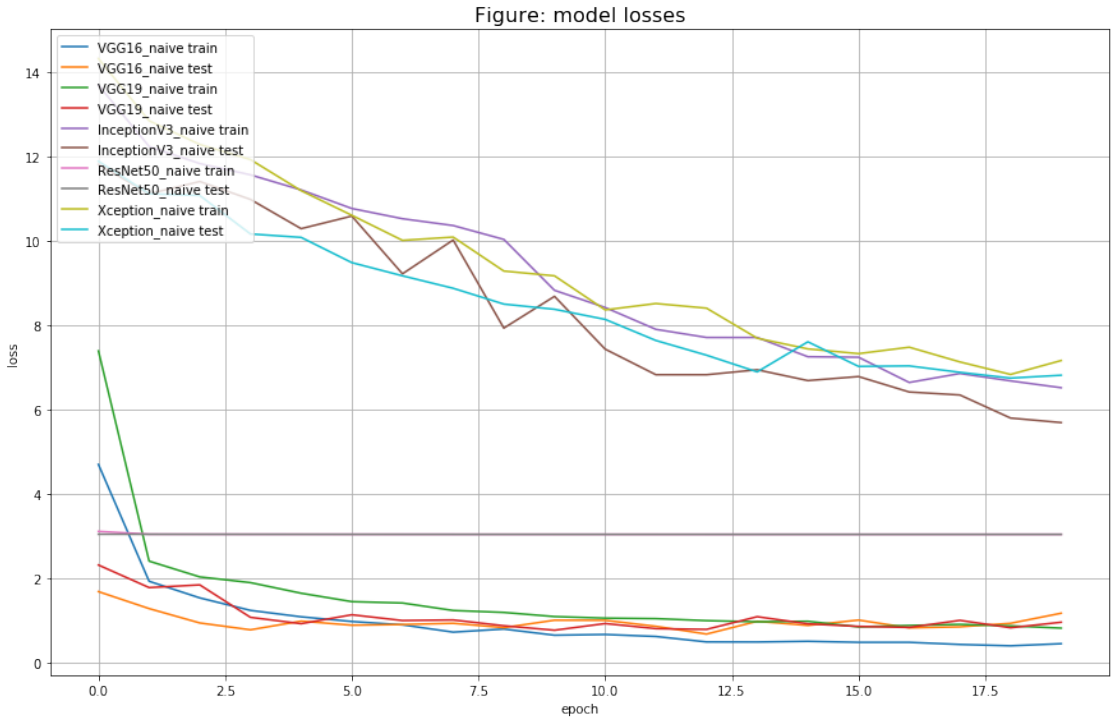
4.7.1 Primeras conclusiones

La primera conclusión clara es que la ResNet50 queda descartada.

Para tener una visión más clara podemos dibujar conjuntamente las métricas anteriores.

```
plot_histories(['VGG16_naive','VGG19_naive','InceptionV3_naive',
                'ResNet50_naive','Xception_naive'])
```





A la vista de esta comparativa, aparentemente la InceptionV3 parece ofrecer peores resulta-

dos que las VGG, que parecen comportarse de modo similar.

Sin embargo, vemos que el poder de clasificación del top model basado en las VGG se estanca entorno a las 10 épocas de entrenamiento, mientras que el basado en la InceptionV3 sigue mejorando a buen ritmo. Algo similar ocurre con la Xception.

No obstante, la red Xception ha generado un top model tan grande (200 MB), que ni siquiera lo puedo subir al repositorio. Por ahora no voy a centrarme en esta red. No la encuentro manejable, pues además es la que peor rendimiento tiene, y por ahora la descarto.

Por lo tanto, voy a volver a entrenar el top model basado en la InceptionV3, pero esta vez con 100 épocas.

4.7.2 Segunda oportunidad a la InceptionV3, con 100 épocas

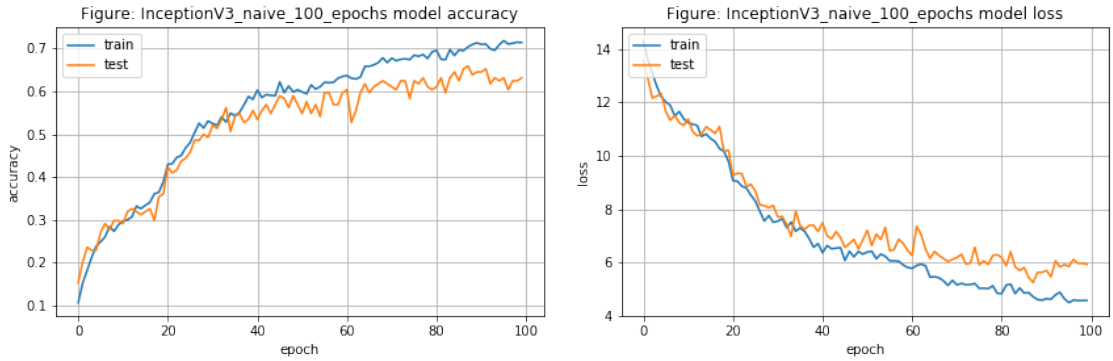
```
input_tensor=Input(shape=(width,height,3))
model = applications.InceptionV3(include_top=False, weights='imagenet',
                                 input_tensor=input_tensor)
model.name = 'InceptionV3'
generate_intermediate_data(model,train_samples,'train')
generate_intermediate_data(model,validation_samples,'validation')

run_experiment('InceptionV3_naive_100_epochs',model,epochs=100)
loss,acc = evaluate_top_model('InceptionV3_naive_100_epochs','InceptionV3')
print 'InceptionV3 100 epochs loss in test data:     %s' % loss
print 'InceptionV3 100 epochs accuracy in test data: %s' % acc
```

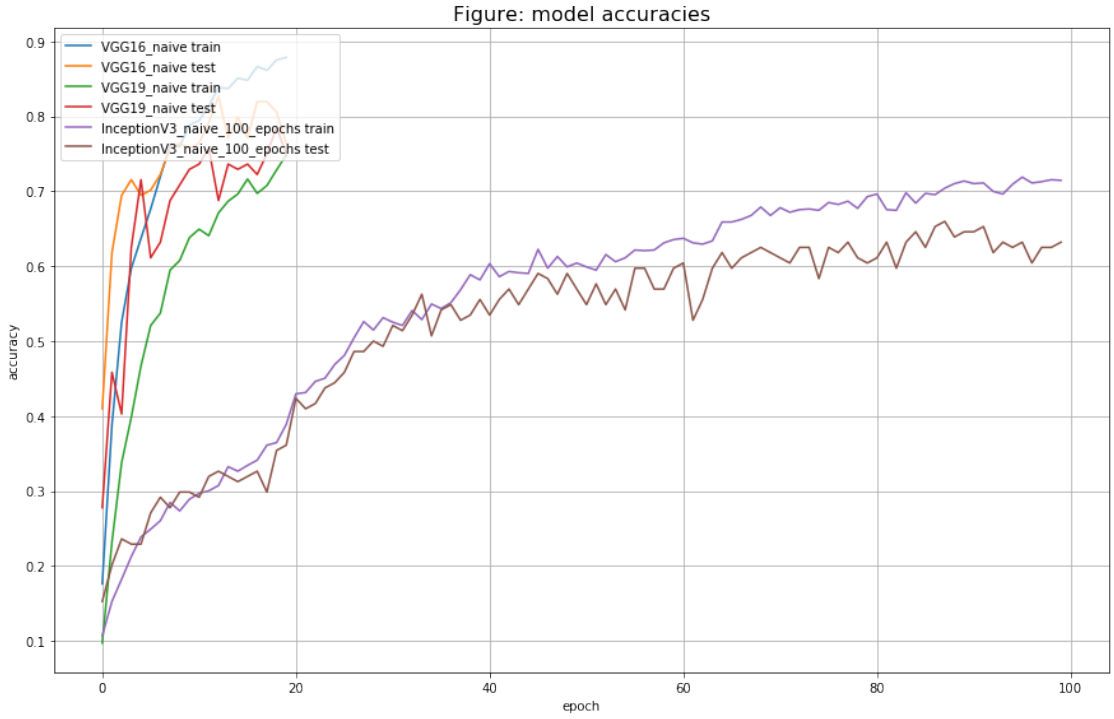
```
intermediate train data for InceptionV3 already generated
intermediate validation data for InceptionV3 already generated
starting InceptionV3_naive_100_epochs experiment
experiment InceptionV3_naive_100_epochs is already done
InceptionV3 100 epochs loss in test data:     5.93235464229
InceptionV3 100 epochs accuracy in test data: 0.631944444444
```

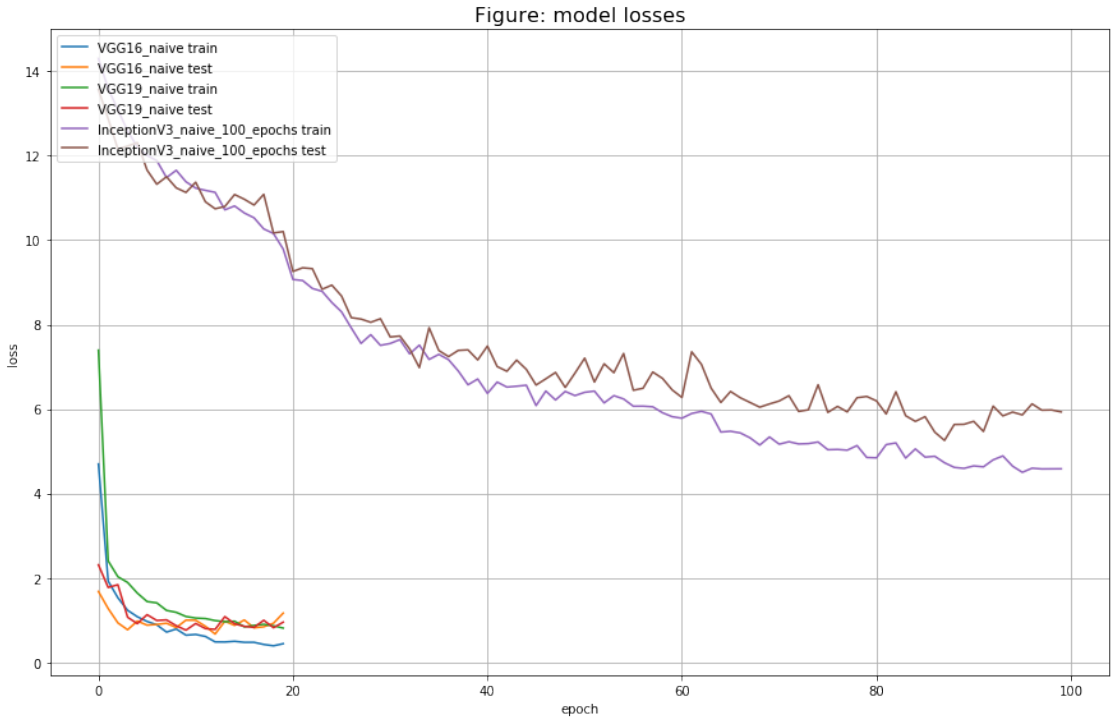
Veamos las gráficas con los resultados:

```
print
plot_history('InceptionV3_naive_100_epochs')
print
plot_histories(['VGG16_naive','VGG19_naive','InceptionV3_naive_100_epochs'])
```

Figure: metrics for InceptionV3_naive_100_epochs







4.8 Explotación de top models basados en la red VGG16

Dado que la VGG16 es la que mejores resultados ofrece en el experimento naive llevado a cabo anteriormente, vamos a seguir probando con esta red, pero utilizando top models distintos.

Se han llevado a cabo los siguientes experimentos, obteniéndose los siguientes resultados:

4.8.1 Primera capa densa con menos neuronas - 64

Obteniéndose peores resultados.

```
loss,acc = evaluate_top_model('VGG16_dense_64','VGG16')
print 'VGG16_dense_64 loss in test data:     %s' % loss
print 'VGG16_dense_64 accuracy in test data: %s' % acc
```

```
VGG16_dense_64 loss in test data:     2.09929230478
VGG16_dense_64 accuracy in test data: 0.347222222222
```

4.8.2 Primera capa densa con más neuronas - 256

Los resultados que se obtienen mejoran bastante a los obtenidos utilizando 128 neuronas. Sin embargo, el tiempo de computación se ha penalizado bastante. Por lo tanto, en los siguietes experimentos voy a seguir con 128 neuronas, que si no esto no acaba nunca.

```
loss,acc = evaluate_top_model('VGG16_dense_256','VGG16')
print 'VGG16_dense_256 loss in test data:     %s' % loss
print 'VGG16_dense_256 accuracy in test data: %s' % acc
```

```
VGG16_dense_256 loss in test data:     0.694945909083
VGG16_dense_256 accuracy in test data: 0.819444444444
```

4.8.3 Activación de la primera capa densa de tipo 'sigmoid'

```
loss,acc = evaluate_top_model('VGG16_dense_sigmoid','VGG16')
print 'VGG16_dense_sigmoid loss in test data:     %s' % loss
print 'VGG16_dense_sigmoid accuracy in test data: %s' % acc
```

```
VGG16_dense_sigmoid loss in test data:     0.656870785687
VGG16_dense_sigmoid accuracy in test data: 0.861111111111
```

Buá!!

Llegamos a un 86% de acierto.

4.8.4 Activación de la primera capa densa de tipo 'tanh'

```
loss,acc = evaluate_top_model('VGG16_dense_tanh','VGG16')
print 'VGG16_dense_tanh loss in test data:     %s' % loss
print 'VGG16_dense_tanh accuracy in test data: %s' % acc
```

```
VGG16_dense_tanh loss in test data:     0.701409419378
VGG16_dense_tanh accuracy in test data: 0.833333333333
```

No está mal, pero no mejora el anterior.

Las siguientes pruebas las hago por lo tanto utilizando el activador sigmoid.

4.8.5 Activador sigmoid y optimizador SGD

```
loss,acc = evaluate_top_model('VGG16_dense_sigmoid_sgd','VGG16')
print 'VGG16_dense_sigmoid_sgd loss in test data:     %s' % loss
print 'VGG16_dense_sigmoid_sgd accuracy in test data: %s' % acc
```

```
VGG16_dense_sigmoid_sgd loss in test data:     0.602853775024
VGG16_dense_sigmoid_sgd accuracy in test data: 0.819444444444
```

No mejora.

4.8.6 Activador sigmoid y optimizador Adagrad

```
loss,acc = evaluate_top_model('VGG16_dense_sigmoid_adagrad','VGG16')
print 'VGG16_dense_sigmoid_adagrad loss in test data:     %s' % loss
print 'VGG16_dense_sigmoid_adagrad accuracy in test data: %s' % acc
```

```
VGG16_dense_sigmoid_adagrad loss in test data:     0.569188520312
VGG16_dense_sigmoid_adagrad accuracy in test data: 0.868055555556
```

Mejora sutilmente, no sustancialmente, el optimizador rmsprop.

Probaremos el optimizador Adam también.

4.8.7 Activador sigmoid y optimizador Adam

```
loss,acc = evaluate_top_model('VGG16_dense_sigmoid_adam','VGG16')
print 'VGG16_dense_sigmoid_adam loss in test data:     %s' % loss
print 'VGG16_dense_sigmoid_adam accuracy in test data: %s' % acc
```

```
VGG16_dense_sigmoid_adam loss in test data:     0.580873813894
VGG16_dense_sigmoid_adam accuracy in test data: 0.8125
```

No aporta mejora alguna.

4.8.8 256 neuronas, activador sigmoid y optimizador Adagrad

Y por último vamos a hacer el experimento final, que lo componemos utilizando el mejor activador y optimizador encontrado en los puntos anteriores, pero esta vez utilizando de nuevo 256 neuronas, aunque el tiempo de cálculo se incremente.

```
loss,acc = evaluate_top_model('VGG16_dense_256_sigmoid_adagrad','VGG16')
print 'VGG16_dense_256_sigmoid_adagrad loss in test data:     %s' % loss
print 'VGG16_dense_256_sigmoid_adagrad accuracy in test data: %s' % acc
```

```
VGG16_dense_256_sigmoid_adagrad loss in test data:     0.5493507021
VGG16_dense_256_sigmoid_adagrad accuracy in test data: 0.833333333333
```

5 Tabla de resultados de todos los experimentos

Red preentrenada	Epochs	Neuronas	Activador	Optimizador	Loss
VGG16	20	128	relu	rmsprop	1.167
VGG19	20	128	relu	rmsprop	0.953
InceptionV3	20	128	relu	rmsprop	5.693
ResNet50	20	128	relu	rmsprop	3.039
Xception	20	128	relu	rmsprop	6.817
InceptionV3	100	128	relu	rmsprop	5.932
VGG16	20	64	relu	rmsprop	2.099
VGG16	20	256	relu	rmsprop	0.694
VGG16	20	128	sigmoid	rmsprop	0.656
VGG16	20	128	tanh	rmsprop	0.701
VGG16	20	128	sigmoid	sgd	0.602
VGG16	20	128	sigmoid	adagrad	0.569
VGG16	20	128	sigmoid	adam	0.580
VGG16	20	256	sigmoid	adagrad	0.549

6 Conclusiones

En esta práctica se han utilizado 5 redes de clasificación de imágenes preentrenadas como herramienta para clasificar un conjunto bastante limitado de imágenes de polen.

La técnica seguida ha consistido en reemplazar las últimas capas de la red preentranada por un modelo que es el que realmente se ha entrenado con nuestros datos de entrenamiento.

Se ha profundizado bastante en encontrar la red que mejores resultados da con un top model dado y se ha trabajado en la búsqueda de los mejores parámetros de entrenamiento del top model. No se ha profundizado en la arquitectura del top model, más hayá del numero de neuronas del mismo o de la función de activación de su primera capa densa.

Se han presentado los resultados de todos los experimentos realizdos y las gráficas de evolución de algunos de los experimentos realizados.

Se ha conseguido clasificar con éxito el 86% de las imágenes de polen de prueba.

7 Referencias

[1] *Ariadne Barbosa Goncalves, Junior Silva Souza, Gercina Goncalves da Silva,Marney Pascoli Cereda, Arnildo Pott, Marco Hiroshi Naka, and HemersonPistori.* Feature extraction and machine learning for the classification ofbrazilian savannah pollen grains.PloS one, 11(6):e0157044, 2016