Classifying images of Colombian celebrities

There are many ways to go about building a CNN. On one hand, you could design and build one from scratch using Keras. At the other end of the spectrum, you could use a service like AWS Rekognition that works entirely behind the scenes and requires almost no input from you. Here, we're going to take the middle ground and use an intermediate tool `fast.ai`. This library has vast functionality and acceptable documentation. To learn more about how to use this library for a variety of deep learning tasks, we recommend their free course Practical Deep Learning for Coders.

This exercise will run very slowly on most laptops, so we're going to run it using the free resources of Google Colab. (It can also work on AWS, but we found this setup to be faster and easier.) We have designed this case so that you can either run the optimization on Colab or simply load trained versions of the CNN to see how they perform.

Notebook setup for Colab

If you have this notebook in your Google Drive, you can open it with Google Colab. Run the following cells to mount your Drive and make its files accessible from within the notebook. You will be prompted to open an URL and retrieve a code, which you'll paste in the field below:

In [1]:
```
# This cell will prompt you to connect this notebook with your google a
ccount.
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
root_dir = "/content/gdrive/My Drive/"
base_dir = root_dir + 'Colab Notebooks/case_18.2'
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?
client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleuser
content.com&redirect_uri=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&scope=emai
l%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdocs.test%20https%3A%2F%2
```

```
t%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdocs.test%20https%3A%2F%2
Fwww.googleapis.com%2Fauth%2Fdrive%20https%3A%2F%2Fwww.googleapis.com%2
Fauth%2Fdrive.photos.readonly%20https%3A%2F%2Fwww.googleapis.com%2Faut
h%2Fpeopleapi.readonly&response_type=code

Enter your authorization code:
··········
Mounted at /content/gdrive
```

In [2]:
```
# Set up Colab to have the ideal settings for fast.ai
!curl -s https://course.fast.ai/setup/colab | bash
```

```
Updating fastai...
Done.
```

Go to `Runtime > Change Runtime Type` and make sure that you have enabled GPU acceleration for this notebook.

Setup for Jupyter Notebooks

Run this cell if operating locally or on AWS as a Jupyter notebook:

In [0]:
```
import os
base_dir = os.getcwd()
```

Setup for everyone

Run these cells from wherever you are operating (Colab or Jupyter):

In [0]:
```
from fastai.vision import *
np.random.seed(42)
```

In [4]:
```
# Define folders for photos and models
```

```
data_path = Path(base_dir + '/photos')
model_path = Path(base_dir + '/models')

# Print for confirmation
data_path
```

Out[4]:

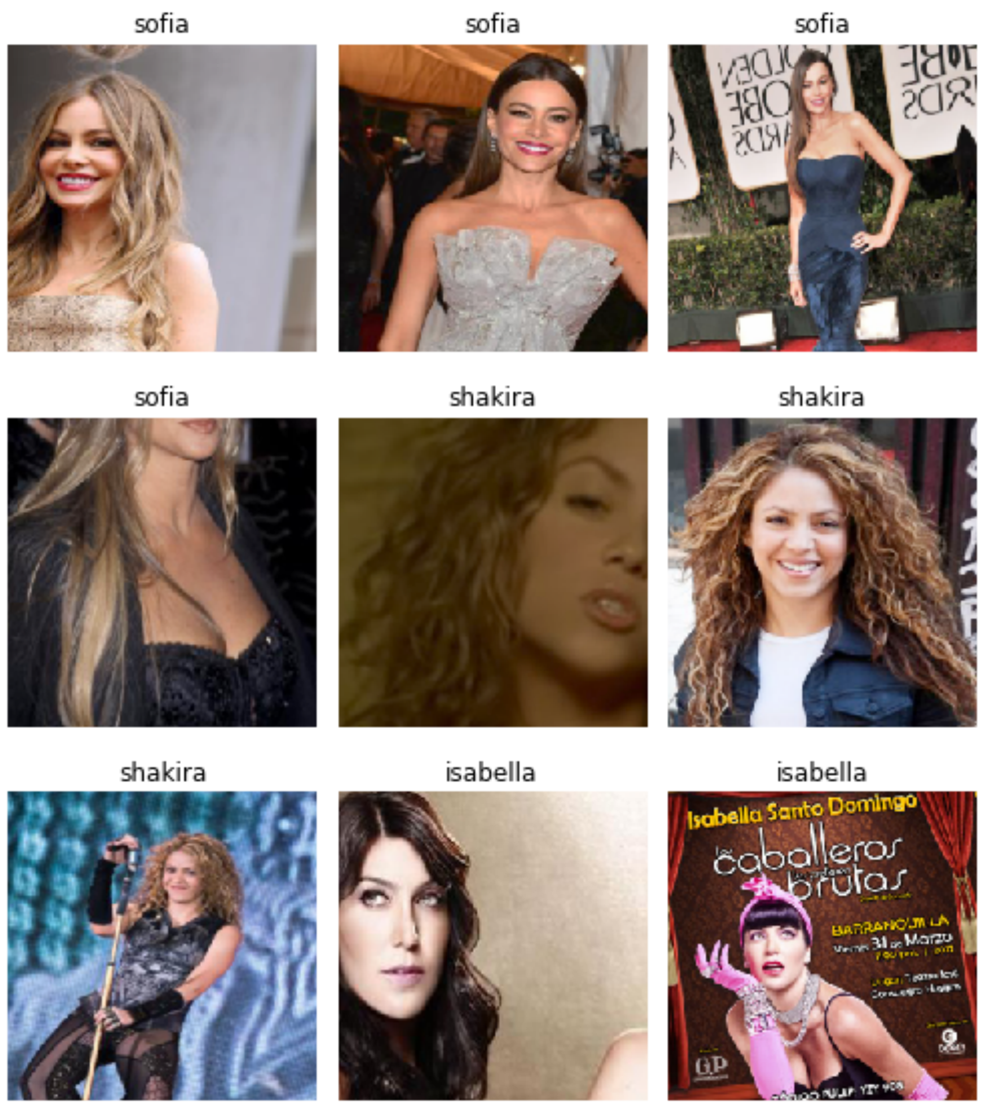
```
PosixPath('/content/gdrive/My Drive/Colab Notebooks/case_18.2/photos')
```

Preparing the data

In `fastai`, we don't feed our data directly to the model. Instead, we package it into an ImageDataBunch object which will contain the data, know any labels for classification, and know which data is in the training/testing/validation sets. There are many ways to populate an ImageDataBunch object.
The following code will populate the ImageDataBunch from our folder structure (docs), apply standard ImageNet image augmentation to our dataset (docs), resize the modified images to a standard size of 224 pixels, and normalize them using the standard ImageNet statistics (docs):

In [6]:

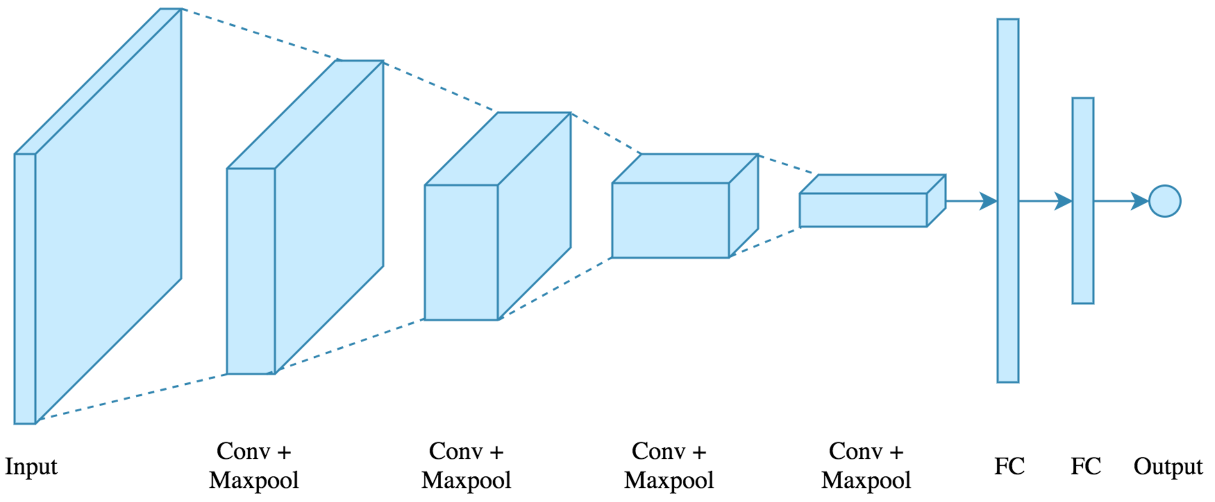
```
# Create a data bunch from the folder of images.
data = ImageDataBunch.from_folder(\
        path=data_path,             # Use the path defined earlier
        ds_tfms=get_transforms(),    # Use standard ImageNet data augme
ntation
        size=224)\
        .normalize(imagenet_stats)   # Use standard ImageNet normalizat
ion

# Show a sample of the pictures
data.show_batch(rows=3, figsize=(7,8))
```



Notice how our images are now all the same size, and labeled according to our folders. Image augmentation involves cropping, and may occasionally cut out important parts of the image. You can sometimes also see mirror padding added to the edges of the images.

Importing a pre-trained model

We'll go more into the structure of CNNs later, but for now, know that CNNs consist of a stack of layers that detect increasingly complex patterns, followed at the end by a handful of fully-connected layers (FCs) that use those patterns to produce a final decision:



Source: [Applied Deep Learning - Part 4: Convolutional Neural Networks]

Today, we'll use [ResNet34], a 34-layer CNN architecture that was the clear state-of-the-art in May of 2016. Training these networks from scratch takes a lot of images and a lot of time, but luckily there's no need to do that. This pre-trained network has already learned to see all sorts of basic and complex patterns. We will re-train only the last layers; i.e. the ones that actually synthesize earlier patterns to get to the final results.

We import the architecture of ResNet34 into a [cnn_learner]. `fastai` keeps all the pre-trained convolutional layers from ResNet34, and then adds a few more layers at the end (one 2D pooling layer, one flattening layer, and blocks of FC layers with batch normalization, dropout, and ReLu activation). All of the pre-trained layers are frozen at the start, so that their weights won't update during training. We'll train the new layers first, since they have been initialized with random weights:

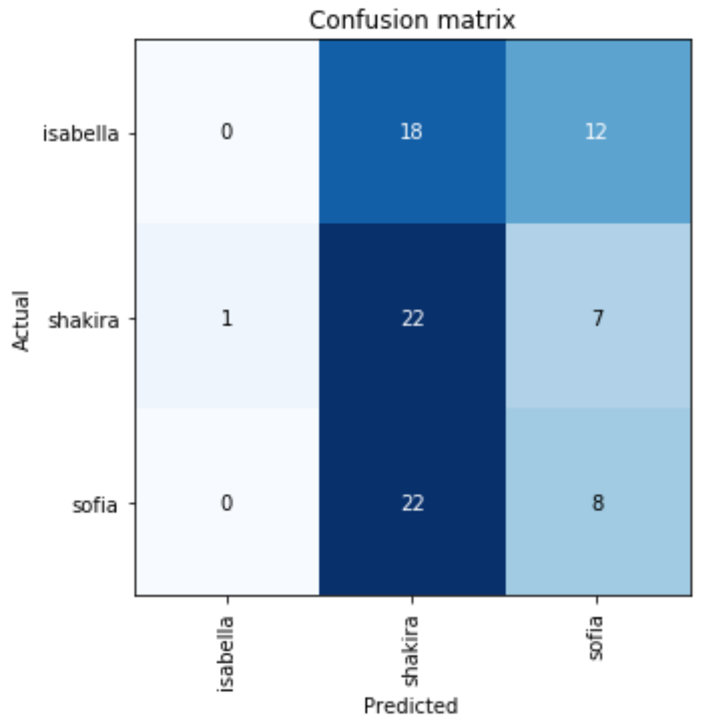
```
In [7]: # Define a new learner, based on the pre-trained ResNet34 CNN

learn = cnn_learner(data=data,                          # Our ImageDataBunch
 from above
                    pretrained=True,
                    base_arch = models.resnet34,  # Base architecture
                    metrics=error_rate);
```

```
Downloading: "https://download.pytorch.org/models/resnet34-333f7ec4.pt
h" to /root/.cache/torch/checkpoints/resnet34-333f7ec4.pth
100%|██████████| 83.3M/83.3M [00:01<00:00, 54.6MB/s]
```

In order to investigate how well our model is doing, we will create a ClassificationInterpretation object from the learner. First, we'll see a confusion matrix, and then we'll use plot_top_losses to see which pictures the model classified most incorrectly:

```
In [9]: # Print out the current state of the model
interp = ClassificationInterpretation.from_learner(learn)
print('Confusion Matrix:')
print(interp.confusion_matrix())
interp.plot_confusion_matrix(figsize=(5,5))
# interp.plot_top_losses(9, figsize=(15,11))
```

```
Confusion Matrix:
[[ 0 18 12]
 [ 1 22  7]
 [ 0 22  8]]
```



For a good working model, the confusion matrix should have dark squares along the main diagonal and lighter squares elsewhere. From this, we can tell that the model corresponding to the above matrix is not doing so well.

Training the new layers

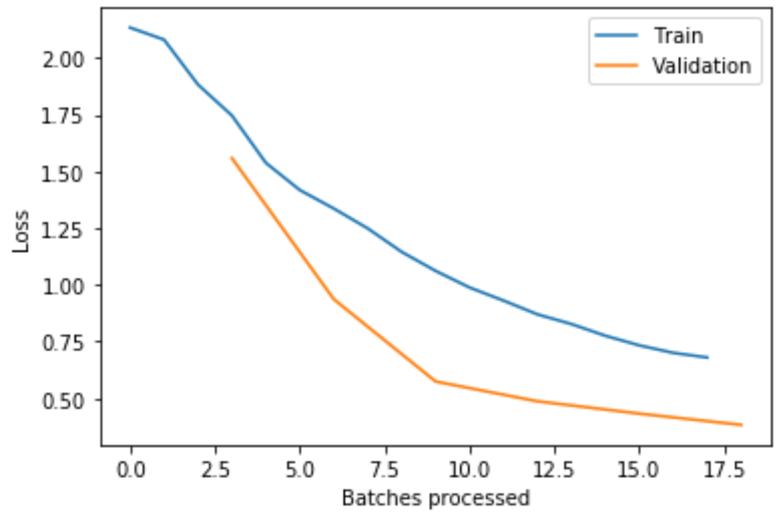
We train the new layers using the 1cycle policy, a method that varies the learning rate so as to achieve super-fast convergence. We train with all the convolutional layers still frozen, updating weights only in the new layers. Each epoch means that the learner trained on all the data, and updated its weights accordingly:

In [0]:
```
learn.fit_one_cycle(6)
```

epoch	train_loss	valid_loss	error_rate	time
0	1.882097	1.557969	0.533333	00:57
1	1.417378	0.936620	0.333333	00:06
2	1.144963	0.573917	0.166667	00:04
3	0.930420	0.486677	0.144444	00:04
4	0.775473	0.432344	0.122222	00:04
5	0.679738	0.382795	0.122222	00:04

Even just training the last few layers, we can bring the classification error rate down from 53% to 12%:

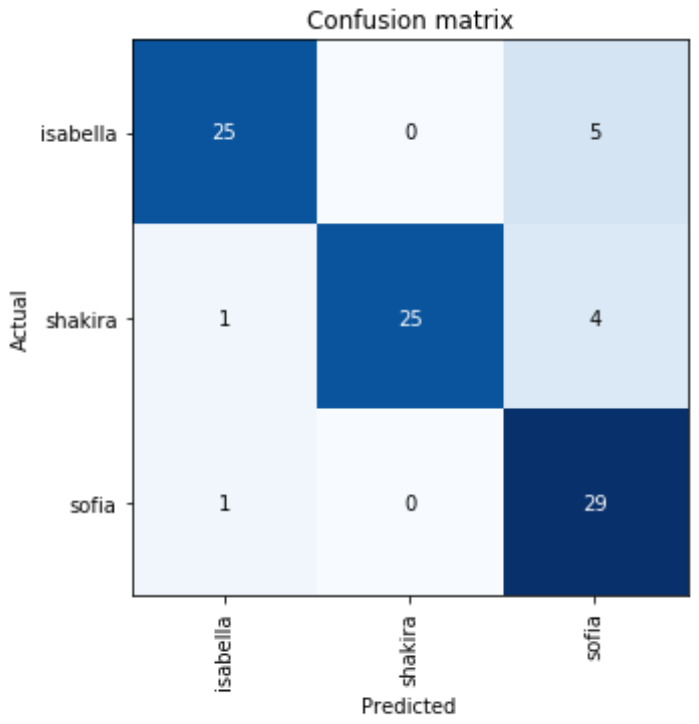
In [0]:
```
learn.recorder.plot_losses()
```

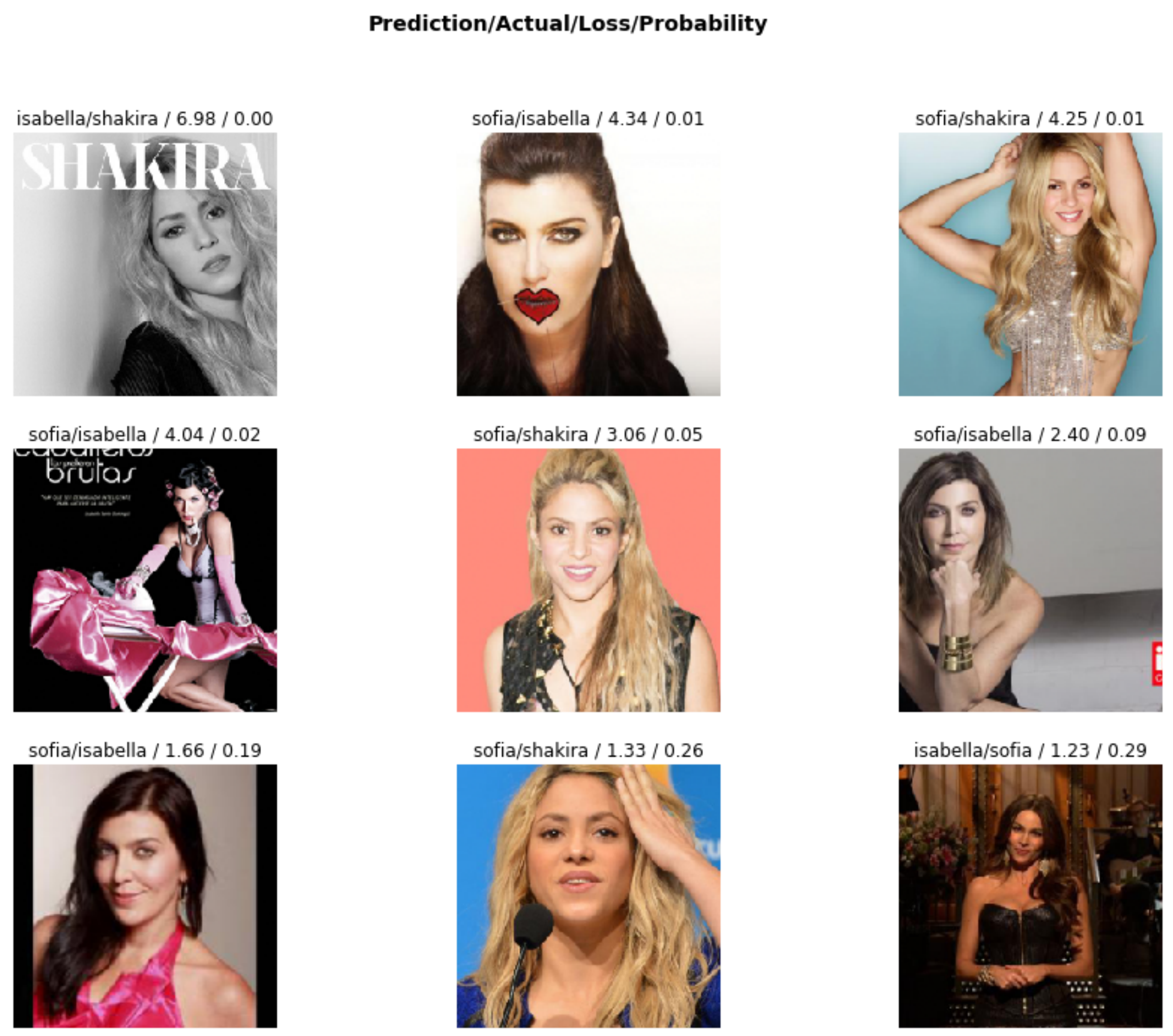


In [0]:
```
# Print out the current state of the model
interp = ClassificationInterpretation.from_learner(learn)
print('Confusion Matrix:')
print(interp.confusion_matrix())
```

```
interp.plot_confusion_matrix(figsize=(5,5))
interp.plot_top_losses(9, figsize=(15,11))
```

```
Confusion Matrix:
[[25  0  5]
 [ 1 25  4]
 [ 1  0 29]]
```




Prediction/Actual/Loss/Probability
isabella/shakira / 6.98 / 0.00
sofia/isabella / 4.34 / 0.01
sofia/shakira / 4.25 / 0.01
sofia/isabella / 4.04 / 0.02
sofia/shakira / 3.06 / 0.05
sofia/isabella / 2.40 / 0.09
sofia/isabella / 1.66 / 0.19
sofia/shakira / 1.33 / 0.26
isabella/sofia / 1.23 / 0.29

This confusion matrix looks MUCH better!

Saving the learner (CAUTION!)

The save learner cell below overrides any existing saved file with the same name. We've provided pre-trained snapshots of the learner, which you may load below. **DON'T** run the `save` cell unless you want to override what we provided:

In [0]:
```
# # Save the trained model
# learn.save(model_path/'stage-1')
```

In [0]:
```
learn.load(model_path/'stage-1');
```

Training the whole CNN

Now that we're done with the above, we can finally unfreeze the whole network to allow for further convergence:

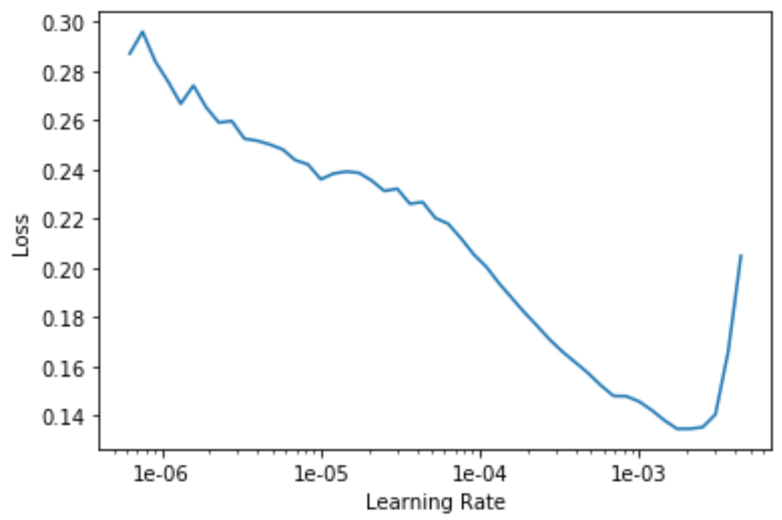
In [0]:
```
# Unfreeze the whole CNN for further training
learn.unfreeze()
```

For training the whole unfrozen model, we're going to use a built-in `fastai` tool for finding the optimal learning rate. This will help our learning progress optimally:

In [0]:
```
# Find the optimal learning rate
learn.lr_find()
```

```
LR Finder is complete, type {learner_name}.recorder.plot() to see the g
raph.
```

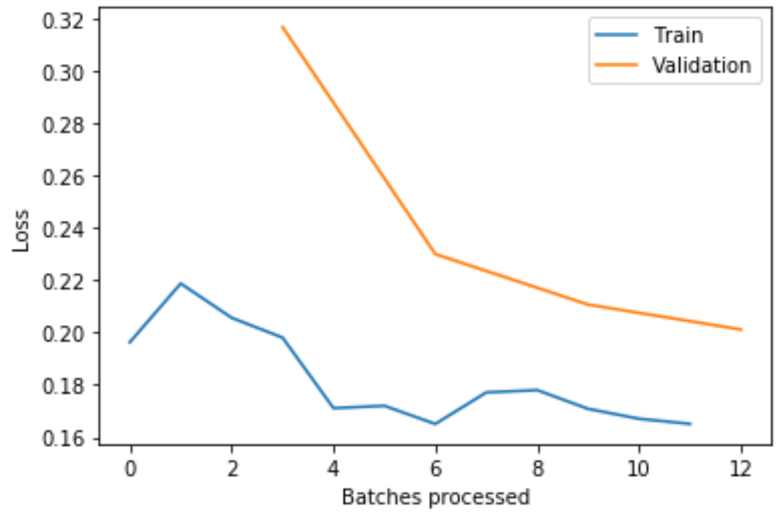
In [0]:
```
# Plot the optimal learning rate
learn.recorder.plot()
```



In [0]:
```
# I choose a range that ends about one order of magnitude before the bo
ttom of the
# valley, and starts one order of magnitude before that.
learn.fit_one_cycle(4, max_lr=slice(2e-5,2e-4))
```

epoch	train_loss	valid_loss	error_rate	time
0	0.205653	0.316800	0.100000	00:04
1	0.171999	0.230005	0.077778	00:04
2	0.177997	0.210711	0.088889	00:04
3	0.165102	0.201188	0.077778	00:04

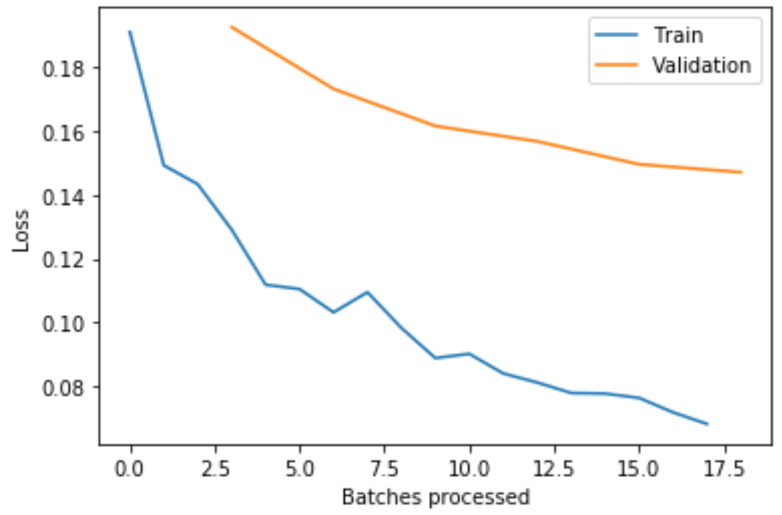
In [0]:
```
learn.recorder.plot_losses()
```



In [0]:
```
# Let's train for a few more epochs
learn.fit_one_cycle(6, max_lr=slice(2e-5,2e-4))
```

epoch	train_loss	valid_loss	error_rate	time
0	0.143370	0.192596	0.088889	00:04
1	0.110487	0.173213	0.088889	00:04
2	0.098306	0.161616	0.066667	00:04
3	0.084039	0.156793	0.055556	00:04
4	0.077723	0.149609	0.055556	00:04
5	0.068209	0.147112	0.055556	00:04

In [0]:
```
learn.recorder.plot_losses()
```



CAUTION! (again!)

The save learner cell below overrides any existing saved file with the same name. We've provided pre-trained snapshots of the learner, which you may load below. **DON'T** run the `save` cell unless you want to override what we provided:

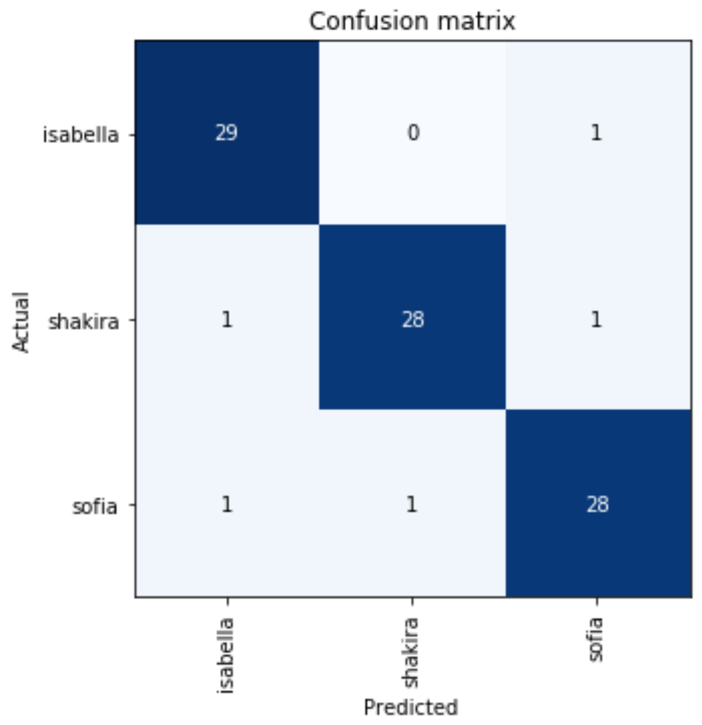
```
In [0]: # # Save the current state of the learner
        # learn.save(model_path/'stage-2')
```

```
In [0]: learn.load(model_path/'stage-2');
```

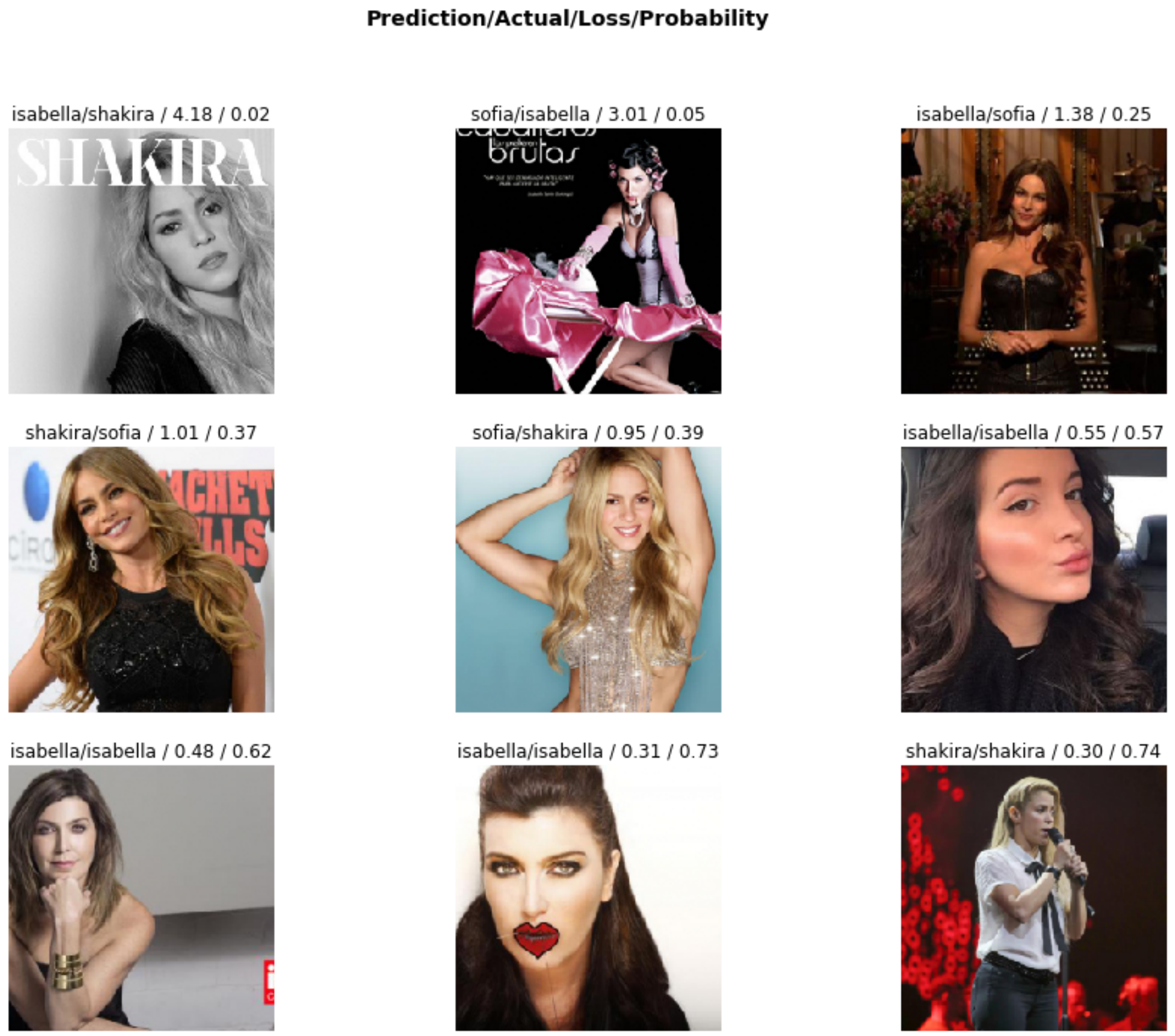
```
In [0]: # Print out the current state of the model
        interp = ClassificationInterpretation.from_learner(learn)
        print('Confusion Matrix:')
        print(interp.confusion_matrix())
        interp.plot_confusion_matrix(figsize=(5,5))
        interp.plot_top_losses(9, figsize=(15,11))
```

```
Confusion Matrix:
[[29  0  1]
```

```
 [ 1 28  1]
 [ 1  1 28]]
```



Confusion matrix
Actual
isabella
shakira
sofia
29
0
1
1
28
1
1
1
28
isabella
shakira
sofia
Predicted


Prediction/Actual/Loss/Probability
isabella/shakira / 4.18 / 0.02
sofia/isabella / 3.01 / 0.05
isabella/sofia / 1.38 / 0.25
shakira/sofia / 1.01 / 0.37
sofia/shakira / 0.95 / 0.39
isabella/isabella / 0.55 / 0.57
isabella/isabella / 0.48 / 0.62
isabella/isabella / 0.31 / 0.73
shakira/shakira / 0.30 / 0.74

Our confusion matrix results are even better now!

Training on far less data

Just to show how much juice you can get out of pre-trained networks, I trained the same network above on just 10 pictures of each celebrity. The results are not quite as good, but still pretty impressive:

train_10