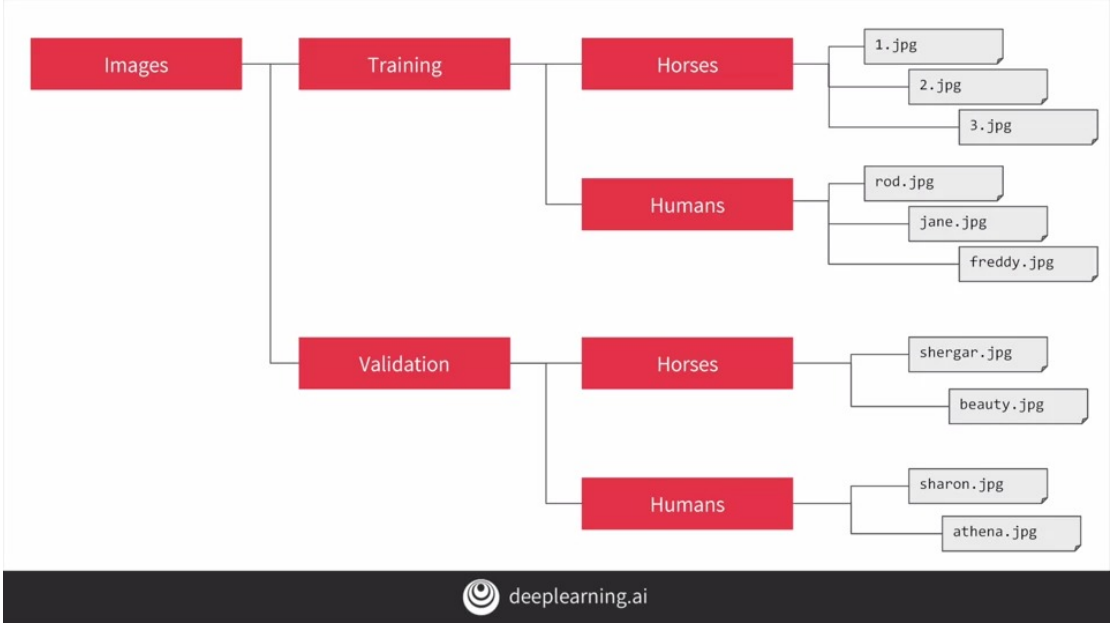
4.1 Understanding ImageGenerator

But what happens when you use larger images and where the feature might be in different locations? For example, how about these images of horses and humans? They have different sizes and different aspect ratios. The subject can be in different locations. In some cases, there may even be multiple subjects.

In addition to that, the earlier examples with a fashion data used a built-in dataset. All of the data was handily split into training and test sets for you and labels were available. In many scenarios, that's not going to be the case and you'll have to do it for yourself.



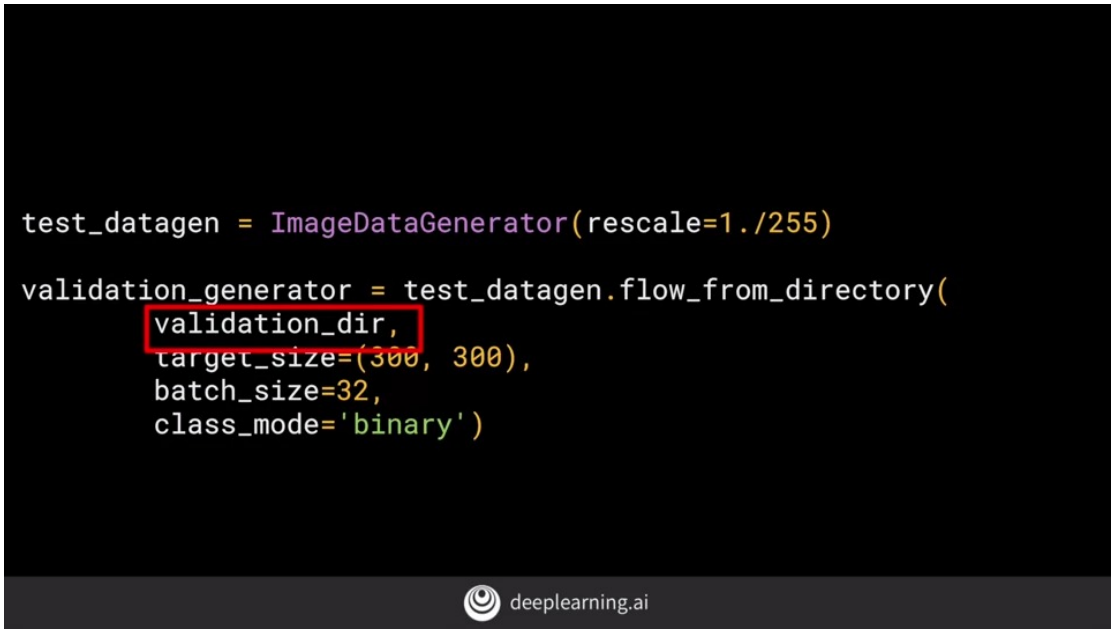
In particular, the **image generator** in TensorFlow.

So for example, consider this directory structure. You have an images directory and in that, you have sub-directories for training and validation. When you put sub-directories in these for horses and humans and store the requisite images in there, the image generator can create a feeder for those images and auto label them for you.

```
from tensorflow.keras.preprocessing.image
import ImageDataGenerator
```

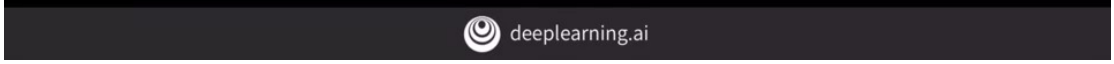
It's a common mistake that people point the generator at the sub-directory. It will fail in that circumstance. **You should always point it at the directory that contains sub-directories that contain your images.**

The nice thing about this code is that the images are resized for you as they're loaded. So you don't need to preprocess thousands of images on your file system.



4.2 Defining a ConvNet to use complex images

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

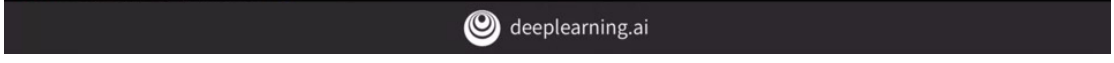


First of all, you'll notice that there are three sets of convolution pooling layers at the top. This reflects the higher complexity and size of the images.

Another thing to pay attention to is the input shape. We resize their images to be 300 by 300 as they were loaded, but they're also color images. So there are three bytes per pixel.

Remember before when you created the output layer, you had one neuron per class, but now there's only one neuron for two classes. That's because we're using a different activation function where sigmoid is great for binary classification, where one class will tend towards zero and the other class tending towards one.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 298, 298, 16)      448
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 149, 149, 16)      0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 147, 147, 32)      4640
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 73, 73, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 71, 71, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 35, 35, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 78400)             0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               40141312
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 40,165,409
Trainable params: 40,165,409
Non-trainable params: 0
```



4.3 Training the ConvNet with fit_generator

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['acc'])
```

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=8,
      epochs=15,
      validation_data=validation_generator,
      validation_steps=8,
      verbose=2)
```

And the verbose parameter specifies how much to display while training is going on. With verbose set to 2, we'll get a little less animation hiding the epoch progress.

```
import numpy as np
from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for fn in uploaded.keys():

  # predicting images
  path = '/content/' + fn
  img = image.load_img(path, target_size=(300, 300))
  x = image.img_to_array(img)
  x = np.expand_dims(x, axis=0)

  images = np.vstack([x])
  classes = model.predict(images, batch_size=10)
  print(classes[0])
  if classes[0]>0.5:
    print(fn + " is a human")
  else:
    print(fn + " is a horse")
```

4.4 Walking through developing a ConvNet

Now that you've learned how to download and process the horses and humans dataset, you're ready to train. When you defined the model, you saw that you were using a new loss function called 'Binary Crossentropy', and a new optimizer called RMSProp. If you want to learn more about the type of binary classification we are doing here, check out this great video from Andrew!

4.5 Walking through training the ConvNet with fit_generator

Course 1 - Part 8 - Lesson 2 - Notebook

```
from tensorflow.keras.optimizers import RMSprop 报错 改成
from tensorflow.python.keras.optimizers import RMSprop
```

```
Each epoch is loading the data, calculating the convolutions and then trying
to match the convolutions to labels. As you can see, the accuracy mostly
increases but it will occasionally deep, showing the gradient ascent of the
learning actually in action. It's always a good idea to keep an eye on
fluctuations in this figure. And if there are too wild, you can adjust the
learning rate.
```

4.6 Adding automatic validation to test accuracy

Course 1 - Part 8 - Lesson 3 - Notebook

In this video, you'll see how you can build validation into the training loop by specifying a set of validation images, and then have TensorFlow do the heavy lifting of measuring its effectiveness with that same.

As well as an image generator for the training data, we now create a second one for the validation data. It's pretty much the same flow. We create a validation generator as an instance of image generator, re-scale it to normalize, and then pointed at the validation directory.

Now, at the end of every epoch as well as reporting the loss and accuracy on the training, it also checks the validation set to give us loss in accuracy there. As the epochs progress, you should see them steadily increasing with the validation accuracy being slightly less than the training.

4.7 Exploring the impact of compressing images

Note that we've changed the input shape to be 150 by 150, and we've removed the fourth and fifth convolutional max pool combinations. Our model summary now shows the layer starting with the 148 by 148, that was the result of convolving the 150 by 150. We'll see that at the end, we end up with a 17 by 17 by the time we're through all of the convolutions and pooling.