A^3 - Aprendizagem Automática Avançada

Advanced

Convolutional Neural Networks

$\mathbb{G}$. Marques

Popular CNN Architectures

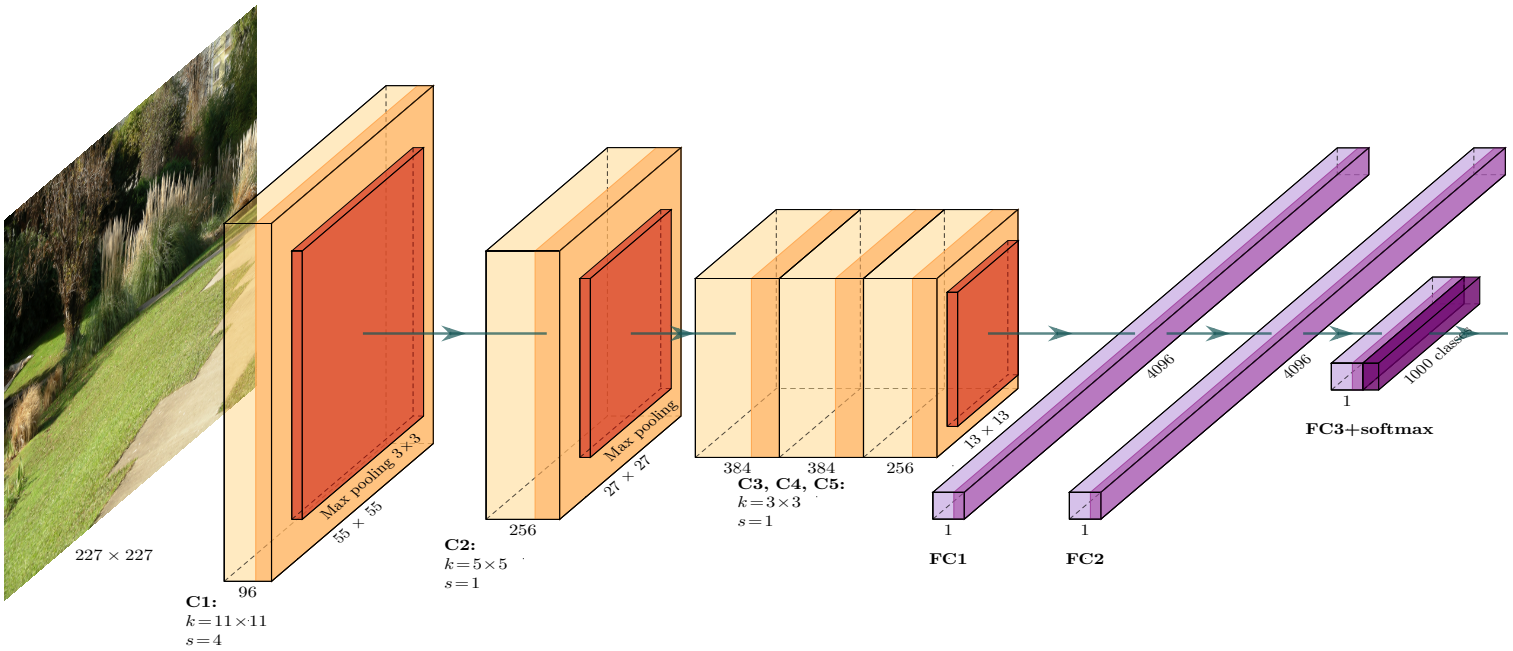
- Over the years, variants of the LeNet5 CNN architecture have been developed and have contributed to significant advances in the field.
- A good measure of the progress made by CNNs, can be seen in the results of competitions and challenges provided by on-line communities in the least few years.
- One particularly relevant challenge is the ILSVRC ImageNet. The ImageNet challenge is divided into several task, like image classification, image classification and localization, and others.
- The winning entries (or runners-up) of the ImageNet classification task are a good barometer of the evolutions made in the field of computer vision by convolutional neural networks. In this competition, the top-five error rate[1] has dropped from 26% (2011) to less than 2% in a period of eight years. Bear in mind that in this task there are 1000 classes, some of which are very subtle (*e.g.* 120 dog breeds), and many images have clutter objects and deformations.
- In the next slides we will address some of the most popular architectures. Most of these models come with Keras along with pre-trained weights on ImageNet.

[1] Top-five error rate is the percentage of the test images that the system did not include the correct answer in the top five predictions.

Popular CNN Architectures

AlexNet:

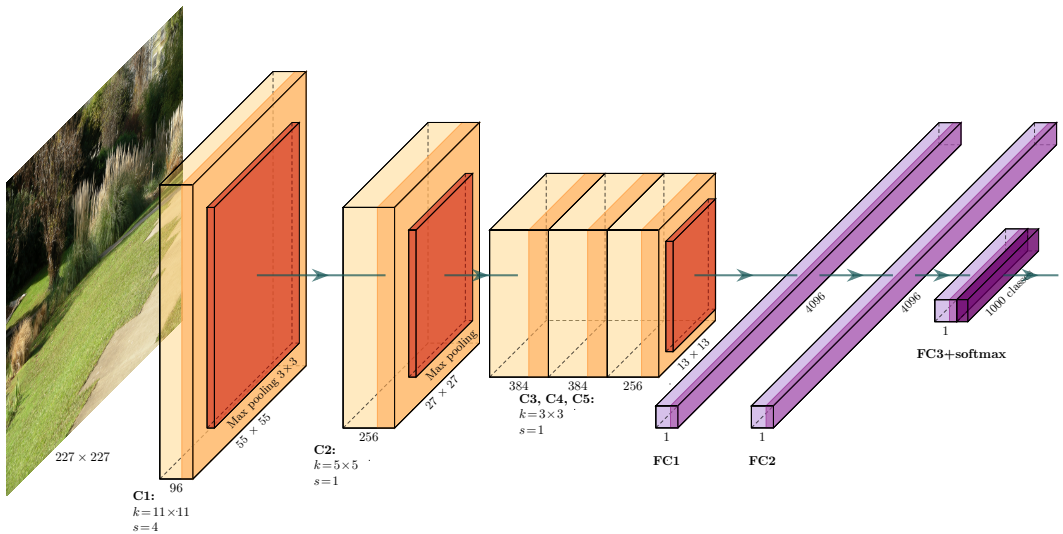
[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks.
In *Advances in neural information processing systems*, pages 1097–1105, 2012



Popular CNN Architectures

AlexNet:

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks.
In *Advances in neural information processing systems*, pages 1097–1105, 2012



- Network composed o eight layers with weights.
- Number of network trainable parameters (over 40 million weights):

C1:	$96+96\times11\times11\times3=34\,944$
C2:	$256+256\times5\times5\times96=614\,656$
C3:	$384+384\times3\times3\times256=885\,120$
C4:	$384+384\times3\times3\times384=1\,327\,488$
C5:	$256+256\times3\times3\times384=884\,992$

Convolutional layers (total: 3 747 200)

FC1:	$4096+4096\times4096=16\,781\,312$
FC2:	$4096+4096\times4096=16\,781\,312$
FC3:	$1000+1000\times4096=4\,097\,000$

Fully connected layers (total: 37 659 624)

Popular CNN Architectures

AlexNet:

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks.
In *Advances in neural information processing systems*, pages 1097–1105, 2012

This is the first Deep Convolutional Neural Network that won the ImageNet ILSVRC 2012 challenge by a large margin (top-five error rate of 17% compared to 26% of the runner-up).

In [1], the authors presented some training methodologies and network configurations that are now standard usage. Compared the LeNet5, this network architecture is similar but wider and deeper.

The chosen non-linearities are the ReLu activations instead hyperbolic tangents, and this network was the first to stack two convolutional layers directly one on top of the other.

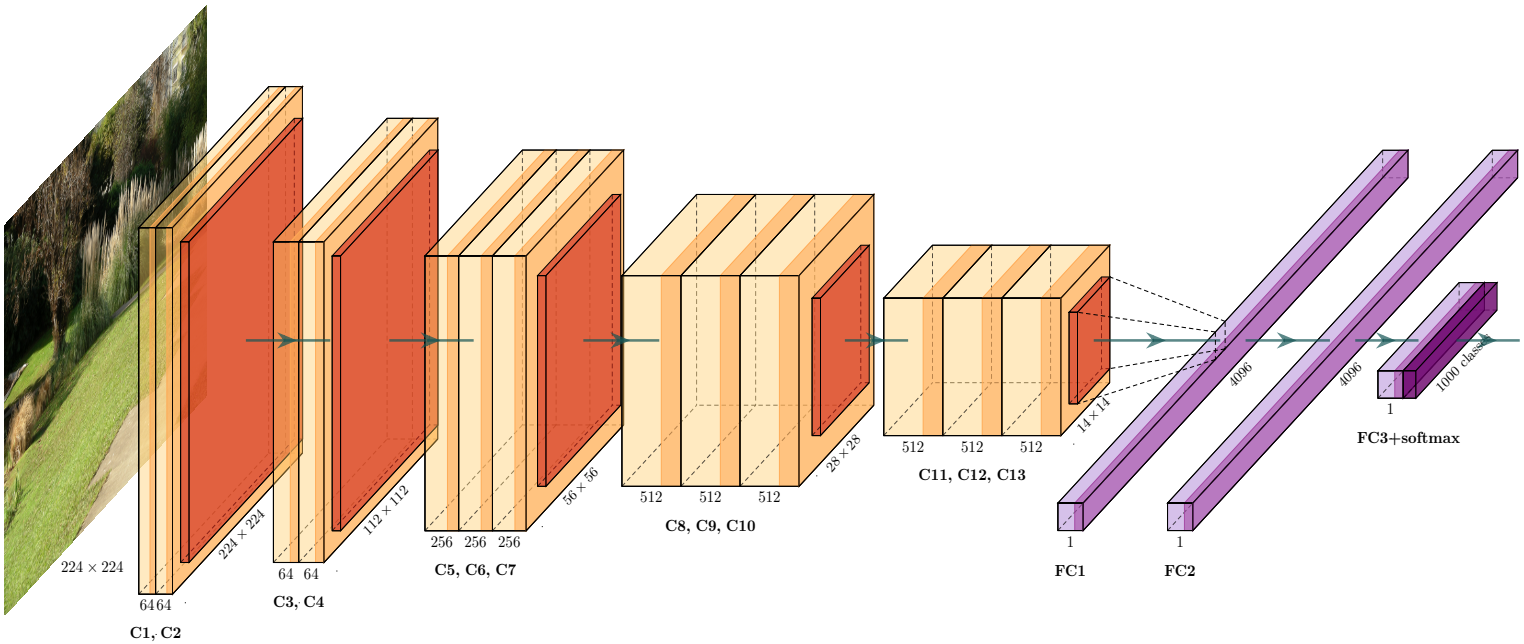
Furthermore, during the training and in order to reduce over-fitting, a dropout rate of 50% was applied to the top two fully-connected layers, and ***data augmentation*** procedure was used to artificially increase the number of training examples.

AlexNet also applies a processing step after the first and the second convolution layers, known as ***local response normalization***, where the strongest activated neurons inhibit other neurons located in the same position in neighboring feature maps (such type of competitive activations have been observed in biological neurons).

Popular CNN Architectures

VGGNet:

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition.
In *arXiv preprint arXiv:1409.1556*, 2014



VGG16

Popular CNN Architectures

VGGNet:

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition.
In *arXiv preprint arXiv:1409.1556*, 2014

The VGG (Visual Geometry Group - Oxford University) was the runner-up in the ImageNet ILSVRC 2014 challenge. This network has a classical architecture with two or three convolutional layers followed by a pooling layer, and again two or three convolutional layers and an pooling layer, and so on (reaching a total of 16 or 19 layers with weighs). All the convolutional filters were composed of 3×3 kernels.

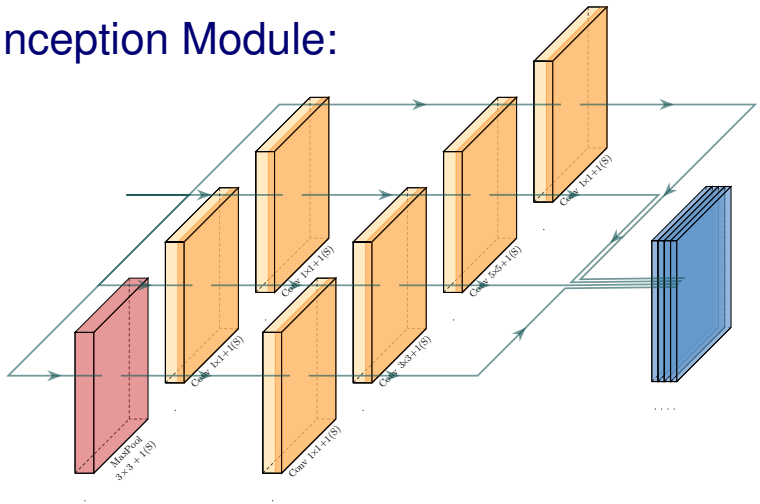
Popular CNN Architectures

GoogLeNet:

[3] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015

The GoogLeNet was the winner in the ImageNet ILSVRC 2014 challenge with a top-five error rate of 7% (its name was in honer of Yan LeCun and his LeNet5 network). This networks is much deeper than previous CNNs, and uses sub-network modules called ***inception modules***. Despite it depth, GoogLeNet makes a more efficient use of the network parameters (*e.g.* GoogLeNet has approximately 10 times less weights than AlexNet).

Inception Module:



Popular CNN Architectures

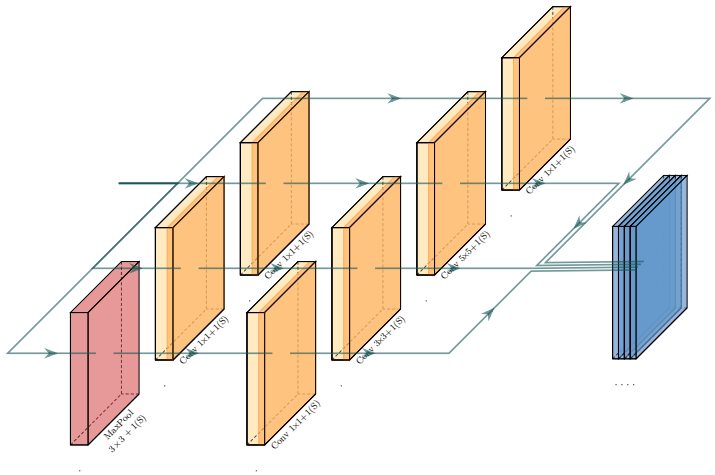
GoogLeNet:

[3] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015

Inception Module:

The input image or feature map is fed to two stacks of convolution layers. The notation 3×3+1(S) means that the layer uses a 3×3 kernel, stride 1 and the same padding. All the convolutional layers have ReLu activations. All the layers use the same padding, meaning that the outputs have the same height and width as the input signal. This allows the four output feature maps of the second stack of convolutional layers to be concatenated along the depth dimension.

- First set of layers:
  - Convolutional layer 1×1+1(S)
  - Convolutional layer 1×1+1(S)
  - MaxPool layer 3×3+1(S)
  - Direct connection to the second set of layers.
- Second set of layers:
  - Convolutional layer 1×1+1(S)
  - Convolutional layer 3×3+1(S)
  - Convolutional layer 5×5+1(S)
  - Convolutional layer 1×1+1(S)



Popular CNN Architectures

GoogLeNet:

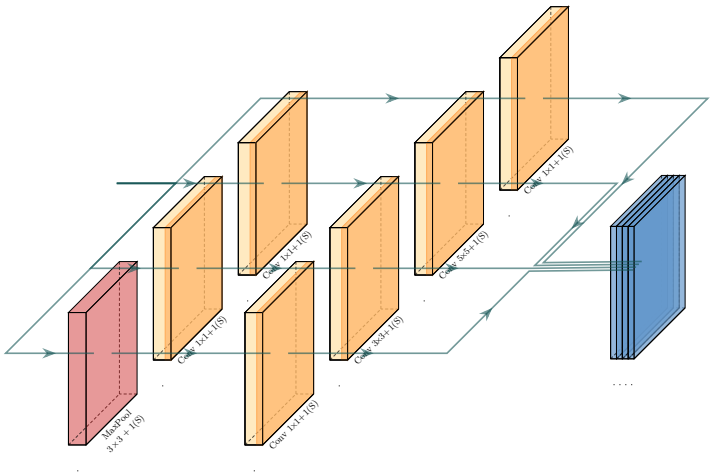
[3] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015

Inception Module:

The input image or feature map is fed to two stacks of convolution layers. The notation 3×3+1(S) means that the layer uses a 3×3 kernel, stride 1 and the same padding. All the convolutional layers have ReLu activations. All the layers use the same padding, meaning that the outputs have the same height and width as the input signal. This allows the four output feature maps of the second stack of convolutional layers to be concatenated along the depth dimension.

- Why 1×1 convolutions?
  - They capture depth patterns
  - They serve as a **bottleneck layer**.
  - Outputs a feature map of depth one.

Note that all paths of the input pass through a 1×1 convolution layer, making it possible to output a feature map of depth four, with the same height and width of the input.



Popular CNN Architectures

GoogLeNet:

[3] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015

The architecture of the GoogLeNet is 22 layers deep (counting only layers with weights). Its first two layers are standard convolutional layers followed by a MaxPool layers. Then there is a tall stack of nine inception modules interleaved by three MaxPool layers. The output layer is a fully-connected, 1000-outputs, softmax unit.

Layer	Type	Kernel Dim.	Output Size
	Input	-	224×224×3
1	Conv.	7×7+2(S)	112×112×64
1.A	MaxPool	3×3+2(S)	56×56×64
2	Conv.	1×1+1(S)	56×56×192
3	Conv.	3×3+1(S)	56×56×192
3.A	MaxPool	3×3+2(S)	28×28×192
4,5	Inception	-	28×28×256
6,7	Inception	-	28×28×480
7.A	MaxPool	3×3+2(S)	14×14×480
8,9	Inception	-	14×14×512
10,11	Inception	-	14×14×512
12,13	Inception	-	14×14×512
14,15	Inception	-	14×14×528

Layer	Type	Kernel Dim.	Output Size
16,17	Inception	-	14×14×832
17.A	MaxPool	3×3+2(S)	7×7×832
18,19	Inception	-	7×7×832
20,21	Inception	-	7×7×1024
21.A	AvPool	7×7+1(V)	1×1×1024
22.A	DropOut (40%)	-	1×1×1024
22	Softmax	-	1×1×1000

Popular CNN Architectures

ResNet:

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.
In *arXiv preprint arXiv:1512.03385*, 2015

Xception:

[5] François Chollet. Xception: Deep learning with depthwise separable convolutions.
In *arXiv preprint arXiv:1610.02357*, 2016

MobilNet:

[6] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications.
In *arXiv preprints arXiv:1704.04861*, 2017

DenseNet:

[7] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks.
CoRR, 2016

Popular CNN Architectures

NasNet:

[8] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition.
CoRR, 2017

EfficientNet:

[9] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks.
CoRR, 2019

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `InceptionResNetV2` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.inception_resnet_v2 import InceptionResNetV2
CNN=InceptionResNetV2(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.inception_resnet_v2 import decode_predictions
from keras.applications.inception_resnet_v2 import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

NOTE: The image (`I: numpy` array), must have been previously resized to fit the specifications of this network ($299 \times 299 \times 3$).

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `InceptionResNetV2` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.inception_resnet_v2 import InceptionResNetV2
CNN=InceptionResNetV2(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.inception_resnet_v2 import decode_predictions
from keras.applications.inception_resnet_v2 import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

```
Predicted:  [[('n02408429', 'water_buffalo',
0.79444367), ('n02504013', 'Indian_elephant',
0.03795137), ('n01871265', 'tusker',
0.01930371), ('n02504458', 'African_elephant',
0.019026486), ('n02403003', 'ox',
0.015965404)]]
```

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `InceptionResNetV2` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.inception_resnet_v2 import InceptionResNetV2
CNN=InceptionResNetV2(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.inception_resnet_v2 import decode_predictions
from keras.applications.inception_resnet_v2 import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

```
Predicted:  [[('n02116738',
'African_hunting_dog', 0.8939504), ('n02117135',
'hyena', 0.015025066), ('n02115913', 'dhole',
0.004112391), ('n02025239', 'ruddy_turnstone',
0.0014297505), ('n02114712', 'red_wolf',
0.00060149544)]]
```

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `InceptionResNetV2` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.inception_resnet_v2 import InceptionResNetV2
CNN=InceptionResNetV2(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.inception_resnet_v2 import decode_predictions
from keras.applications.inception_resnet_v2 import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

```
Predicted:  [[('n02089973',
'English_foxhound', 0.8758193), ('n02100236',
'German_short-haired_pointer', 0.027028315),
('n02412080', 'ram', 0.010217963),
('n02102177', 'Welsh_springer_spaniel',
0.0066568702), ('n02088238', 'basset',
0.0053563244)]]
```

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `InceptionResNetV2` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.inception_resnet_v2 import InceptionResNetV2
CNN=InceptionResNetV2(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.inception_resnet_v2 import decode_predictions
from keras.applications.inception_resnet_v2 import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

```
Predicted:  [[('n02165456', 'ladybug',
0.9162143), ('n02169497', 'leaf_beetle',
0.02738419), ('n07730033', 'cardoon',
0.00051306805), ('n02219486', 'ant',
0.00037433335), ('n11939491', 'daisy',
0.00030720327)]]
```

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `Xception` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.xception import Xception
CNN=Xception(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.xception import decode_predictions
from keras.applications.xception import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

Again, the image size is $(299 \times 299 \times 3)$.

```
Predicted:  [[('n02408429', 'water_buffalo',
0.63482267), ('n02403003', 'ox', 0.2697628),
('n02410509', 'bison', 0.02019861),
('n03868242', 'oxcart', 0.0063165333),
('n01871265', 'tusker', 0.0028641545)]]
```

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `Xception` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.xception import Xception
CNN=Xception(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.xception import decode_predictions
from keras.applications.xception import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

```
Predicted:  [[('n02116738',
'African_hunting_dog', 0.9032454), ('n02117135',
'hyena', 0.020238616), ('n02114712',
'red_wolf', 0.0022175272), ('n02115913',
'dhole', 0.0019797664), ('n02091134',
'whippet', 0.00066944776)]]
```

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `Xception` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.xception import Xception
CNN=Xception(weights='imagenet')
```

The next few lines classify an image.

```
from keras.applications.xception import decode_predictions
from keras.applications.xception import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

```
Predicted:  [[('n02089973',
'English_foxhound', 0.15721874), ('n02101388',
'Brittany_spaniel', 0.14868997), ('n02102177',
'Welsh_springer_spaniel', 0.06549238),
('n02412080', 'ram', 0.057797015),
('n02089867', 'Walker_hound', 0.04349323)]]
```

Using Pre-trained Models from Keras

There are several pre-trained models available in Keras (see Keras Applications). These can be loaded very easily with just a few lines of code. For example, the next lines load the `Xception` model with weights pre-trained on ImageNet (1000 output classes).

```
from keras.applications.xception import Xception
CNN=Xception(weights='imagenet')
```

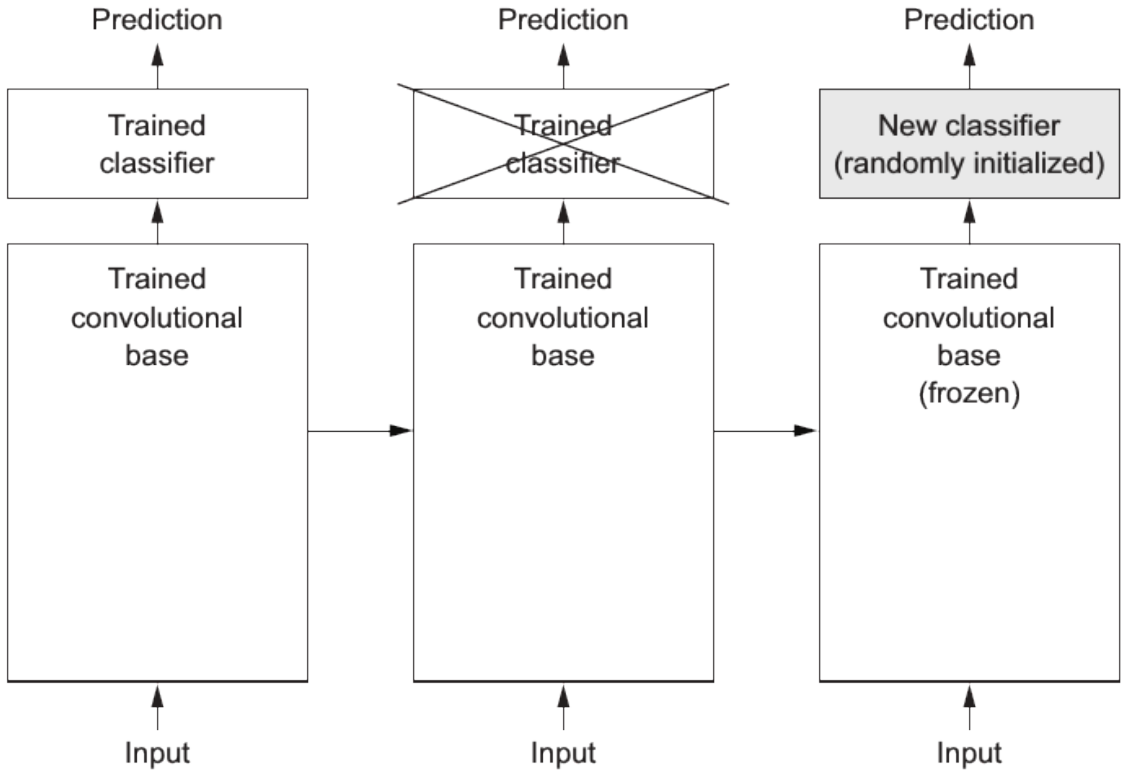
The next few lines classify an image.

```
from keras.applications.xception import decode_predictions
from keras.applications.xception import preprocess_input
I=np.expand_dims(I,axis=0) # add extra dim.
I=preprocess(I)
preds=CNN.predict(I)
print('Predicted:',decode_predictions(preds,top=5))
```

```
Predicted:  [[('n02165456', 'ladybug',
0.96495414), ('n02169497', 'leaf_beetle',
0.016234199), ('n12620546', 'hip',
0.000731681), ('n02219486', 'ant',
0.00064394134), ('n01776313', 'tick',
0.0003810699)]]
```

Transfer Learning

One can build an image classifier with little training data with the help of pre-trained networks. CNNs are comprised of two parts: they start with a series of convolutional and MaxPooling layers (a **convolutional base**) and end with a fully connected MLP network. Transfer learning is a method that uses pre-trained networks and adapts them to perform new classification tasks. The standard approach is to freeze the convolutional base, remove the top MLP network and replace it with another suited for the new task. This new MLP network is then trained with the new set of (few) examples.



by François Chollet

Transfer Learning

One can build an image classifier with little training data with the help of pre-trained networks. CNNs are comprised of two parts: they start with a series of convolutional and MaxPooling layers (a **convolutional base**) and end with a fully connected MLP network. Transfer learning is a method that uses pre-trained networks and adapts them to perform new classification tasks. The standard approach is to freeze the convolutional base, remove the top MLP network and replace it with another suited for the new task. This new MLP network is then trained with the new set of (few) examples.

Tips and Tricks:

- Why does re-using the convolutional base work?
  The maps of pre-trained CNNs (especially with large image databases such as ImageNet) capture generic concepts of real images and therefore are likely to be very useful to the computer vision task at hand. Note that the level of generality depends on the depth of the convolutional layer in the model. The first convolutional layers tend to extract very generic features such as edge or color detectors, whereas the top convolutional layers tend to gather more abstract concepts such as “dog”, “face”, “written text”, etc.
- If your new data differs significantly from the one that was used to train the model, you may want to consider only the first few convolutional layers instead of the whole convolutional base.

Transfer Learning

One can build an image classifier with little training data with the help of pre-trained networks. CNNs are comprised of two parts: they start with a series of convolutional and MaxPooling layers (a **convolutional base**) and end with a fully connected MLP network. Transfer learning is a method that uses pre-trained networks and adapts them to perform new classification tasks. The standard approach is to freeze the convolutional base, remove the top MLP network and replace it with another suited for the new task. This new MLP network is then trained with the new set of (few) examples.

Tips and Tricks:

- It is good practice to first train the top MLP layers with the weights of the convolutional base frozen, even if you are going to retrain the base model (or parts of it). If you don't do this, the weights in the convolutional base will be modified during training. The MLP layers' weights are initialized randomly, and their errors are propagated back through the network during adaptation: this will destroy what was already learned by the pre-trained convolutional layers.
- Usually, it is too computational demanding to train the whole network, but one can retrain just some of the layers of the base model (usually the top ones).
- Once the MLP has been trained, one can unfreeze the desired convolutional base layers and re-train these along with the top MLP layers.

Transfer Learning Example:

Objective:

Build a model to classify pictures of flowers, reusing a pre-trained VGG16 network.

The Dataset:

We will use the Harvard Dataverse Flowers dataset [10]. This is a typical example of a dataset with not enough data to train a deep-CNN network from scratch. In this case, it is a good idea to reuse the convolutional layers of a pre-trained model.
The dataset is comprised of 3670 examples, divided in five classes
(class name|number of examples):


Daisy|663


Dandelion|898


Roses|641


Sunflowers|699


Tulips|799

- This is dataset from Google available in TensorFlow Keras. You can downloaded it with the following command:

```
tf.keras.utils.get_file(origin="https://storage.googleapis.com/
download.tensorflow.org/example_images/flower_photos.tgz",
fname="flower_photos", untar=True)
```

[10] K Tung. Flowers Dataset, 2020.
URL https://doi.org/10.7910/DVN/1ECTVN

Transfer Learning Example:

Objective:

Build a model to classify pictures of flowers, reusing a pre-trained VGG16 network.

The Dataset:

We will use the Harvard Dataverse Flowers dataset [10]. This is a typical example of a dataset with not enough data to train a deep-CNN network from scratch. In this case, it is a good idea to reuse the convolutional layers of a pre-trained model.
The dataset is comprised of 3670 examples, divided in five classes
(class name|number of examples):


Daisy|663


Dandelion|898


Roses|641


Sunflowers|699


Tulips|799

- For training and testing purposes, the dataset is divided into three subsets: a training set with 50% of the data (1835 photos), a validation set with 20% of the data (735 photos), and a test set with 30% of the data (1100 photos).
- Also, we will store the data in three sub-directories: "`train/`", "`validation/ `", and "`test/'`". This will be useful to pipeline the training process with the Keras `ImageDataGenerator` class.

Transfer Learning Example:

The Convolutional Base:

- Instantiating the VGG16 convolutional base:

```
from keras.applications.vgg16 import VGG16
cnnBase=VGG16(weights="imagenet",include_top=False,
        input_shape=(200,200,3))
```

You can pass three arguments:

`weights`	Dataset used to train the network.
`include_top`	To include (or not) the top part of the network.
`input_shape`	Size of the input images.

The `input_shape` parameter resizes all images to the specified dimensions.

- The previous commands loaded the VGG16 network, trained on the ImageNet data. The top part of the network was excluded, since we will replace it with our own. Furthermore, it is a good idea to freeze the convolutional base weights before training the top part.

```
for layer in cnnBase.layers:
    layer.trainable = False
```

or

```
cnnBase.trainable=False
```

Transfer Learning Example:

The Convolutional Base:

```
cnnBase.summary()
```

```
Model: "vgg16"
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 200, 200, 3)]	0
block1_conv1 (Conv2D)	(None, 200, 200, 64)	1792
block1_conv2 (Conv2D)	(None, 200, 200, 64)	36928
block1_pool (MaxPooling2D)	(None, 100, 100, 64)	0
block2_conv1 (Conv2D)	(None, 100, 100, 128)	73856
block2_conv2 (Conv2D)	(None, 100, 100, 128)	147584
block2_pool (MaxPooling2D)	(None, 50, 50, 128)	0
block3_conv1 (Conv2D)	(None, 50, 50, 256)	295168
block3_conv2 (Conv2D)	(None, 50, 50, 256)	590080
block3_conv3 (Conv2D)	(None, 50, 50, 256)	590080
block3_pool (MaxPooling2D)	(None, 25, 25, 256)	0

Transfer Learning Example:

The Convolutional Base:

```
Model: "vgg16"
.
.
.
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 25, 25, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 25, 25, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 25, 25, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 12, 12, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 6, 6, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

Transfer Learning Example:

The Network:

- Create a sequential model:

```
cnn=keras.models.Sequential()
cnn.add(cnnBase)
cnn.add(keras.layers.AvergePooling2D(pool_size=(2,2),strides=2))
cnn.add(keras.layers.Flatten()
cnn.add(keras.layers.Dense(128,activation="relu"))
cnn.add(keras.layers.Dropout(.4))
cnn.add(keras.layers.Dense(5, activation="softmax"))
```

- The output of the convolutional base layer is a $6 \times 6 \times 512$ feature map. To lighten the computational load, an average pooling layer with a 2×2 kernel and a stride of 2 is added at the output of the base layer. This will reduce the size of the feature maps by a factor of four (to $3 \times 3 \times 512$).
- A flatten layer is used to prepare the data for the next fully-connected layers. These are a 128 unit dense layer (with a Dropout rate of 40%) followed by a 5-unit softmax layer. A dropout layer was used for regularization purposes, and the 5-unit softmax output is for our 5-class problem predictions.

Notes:

Test other configurations, including different types of pooling and dropout rates (including none at all), different number of dense layers and number of units in each one.

Transfer Learning Example:

The Network:

- Create a sequential model:

```
cnn=keras.models.Sequential()
cnn.add(cnnBase)
cnn.add(keras.layers.AvergePooling2D(pool_size=(2,2),strides=2))
cnn.add(keras.layers.Flatten()
cnn.add(keras.layers.Dense(128,activation="relu"))
cnn.add(keras.layers.Dropout(.4))
cnn.add(keras.layers.Dense(5, activation="softmax"))
```

- The output of the convolutional base layer is a $6 \times 6 \times 512$ feature map. To lighten the computational load, an average pooling layer with a 2×2 kernel and a stride of 2 is added at the output of the base layer. This will reduce the size of the feature maps by a factor of four (to $3 \times 3 \times 512$).
- A flatten layer is used to prepare the data for the next fully-connected layers. These are a 128 unit dense layer (with a Dropout rate of 40%) followed by a 5-unit softmax layer. A dropout layer was used for regularization purposes, and the 5-unit softmax output is for our 5-class problem predictions.

Notes:

The number of weights in the two top layers is $3 \times 3 \times 512 \times 128 + 128 + 128 \times 5 + 5 = 590\,597$. This number is small compared to the rest of the 14 million weights of the base layer, but if one is not careful, it can become quite large. For example, if we had not used a pooling layer, and chosen the first fully connected layer with 512 units, the number of weights in the top two layers would have been 9 450 501.

Transfer Learning Example:

Training the Network:

- **Preparing the Data**
  The Keras `ImageDataGenerator` class makes it easy to load images from disk and transform them in various ways. This lets us quickly setup Python generators that can automatically fetch batches of images stored on disk and make them ready for our network.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.applications.vgg16 import preprocess_input
dbDir="./flowers/"
dataGen=ImageDataGenerator(preprocessing_function=preprocess_input)
trainGen=dataGen.flow_from_directory(directory=dbDir+"train/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
validGen=dataGen.flow_from_directory(directory=dbDir+"validation/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
testGen=dataGen.flow_from_directory(directory=dbDir+"test/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
```

In the instantiation of the `ImageDataGenerator`, the preprocessing function `preprocess_input` from the VGG16 module was passed as arguments . This prepares the images the way the network expects.

Transfer Learning Example:

Training the Network:

- **Preparing the Data**
  The Keras `ImageDataGenerator` class makes it easy to load images from disk and transform them in various ways. This lets us quickly setup Python generators that can automatically fetch batches of images stored on disk and make them ready for our network.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.applications.vgg16 import preprocess_input
dbDir="./flowers/"
dataGen=ImageDataGenerator(preprocessing_function=preprocess_input)
trainGen=dataGen.flow_from_directory(directory=dbDir+"train/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
validGen=dataGen.flow_from_directory(directory=dbDir+"validation/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
testGen=dataGen.flow_from_directory(directory=dbDir+"test/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
```

Three data generators were created, one for each set (train, validation, and test). In our case, the generators will fetch batches of 32 images, and output resized $200 \times 200 \times 3$ images along with their classes.

Transfer Learning Example:

Training the Network:

- **Preparing the Data**
  The Keras `ImageDataGenerator` class makes it easy to load images from disk and transform them in various ways. This lets us quickly setup Python generators that can automatically fetch batches of images stored on disk and make them ready for our network.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.applications.vgg16 import preprocess_input
dbDir="./flowers/"
dataGen=ImageDataGenerator(preprocessing_function=preprocess_input)
trainGen=dataGen.flow_from_directory(directory=dbDir+"train/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
validGen=dataGen.flow_from_directory(directory=dbDir+"validation/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
testGen=dataGen.flow_from_directory(directory=dbDir+"test/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
```

Note that generators will keep on fetching data indefinitely: they loop endlessly over the images on the target folders. For example, the next loop circles forever over the training images:

```
for I,y in trainGen:
...
```

Transfer Learning Example:

Training the Network:

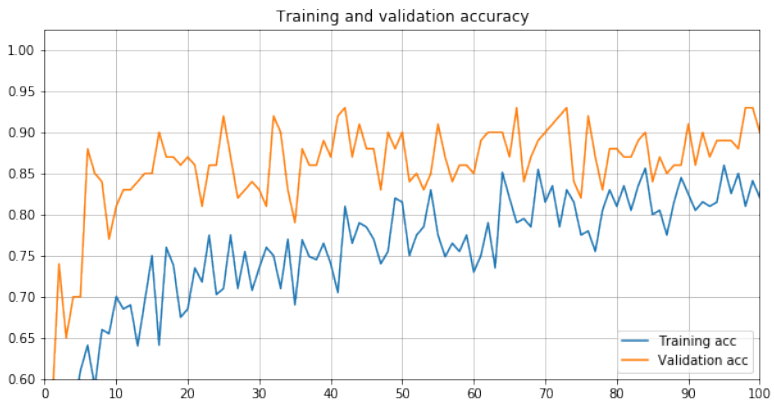
- Choose an optimization method
  `opt = keras.optimizers.Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.99)`
  Attention: the choice of hyper-parameters can influence the final outcome.
- Compile the model
  `cnn.compile(loss="categorical_crossentropy", optimizer=opt,metrics=["accuracy"])`
- Train the model
  `cnn.fit(trainGen,steps_per_epoch=20, validation_data=validGen, validation_steps=10,epochs=100)`

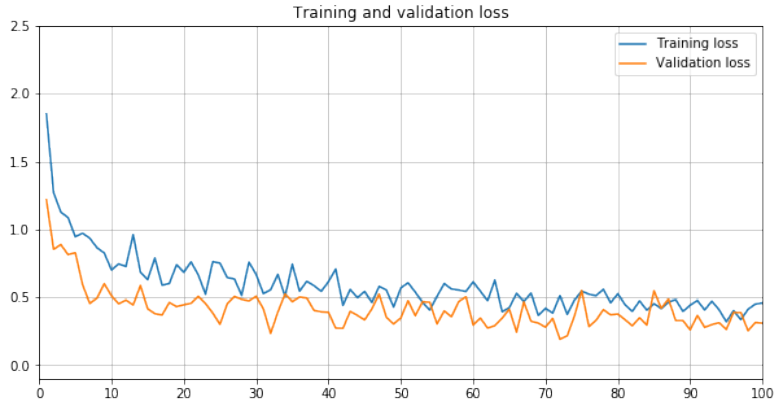
Note that the generators were passed as arguments of the `fit()` function. Since the data is being generated endlessly, the model needs to known how many samples to draw before an epoch is finished. This is the role of the `step_per_epoch` argument. In our case, a number of 20 step per epoch means that the model will draw 20 batches of 32 images each before declaring an epoch over. The `epoch` parameter defines the number of epochs in the training process. The `validation_steps` defines how many batches from the validation generator to draw for evaluation.

Transfer Learning Example:

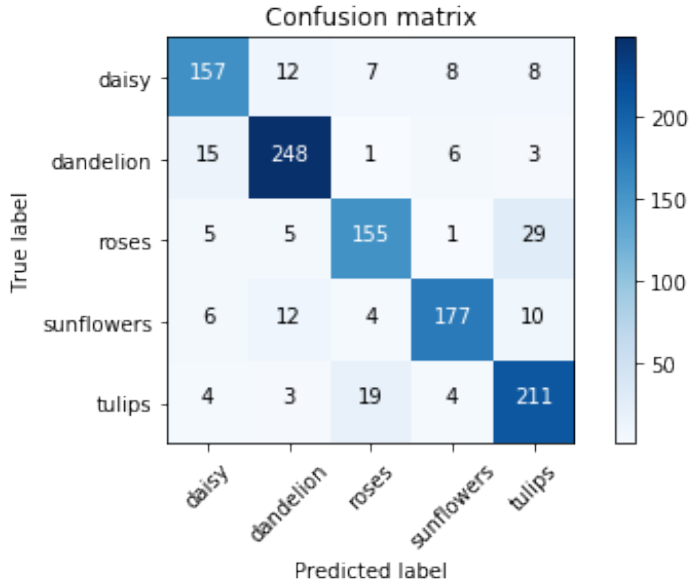
Experimental Results:

- Accuracies and losses during training for the training and validation sets:



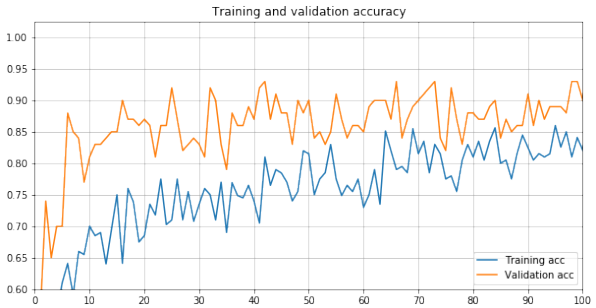


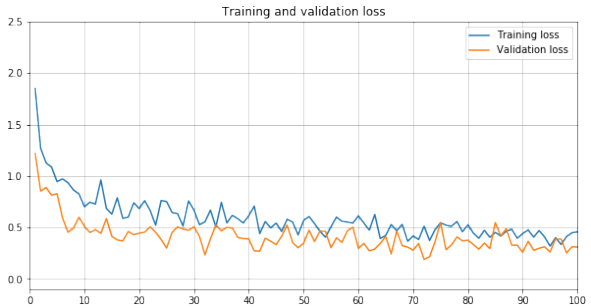
- Test set confusion matrix (accuracy 85.41% - 162 error in 1100 test examples):

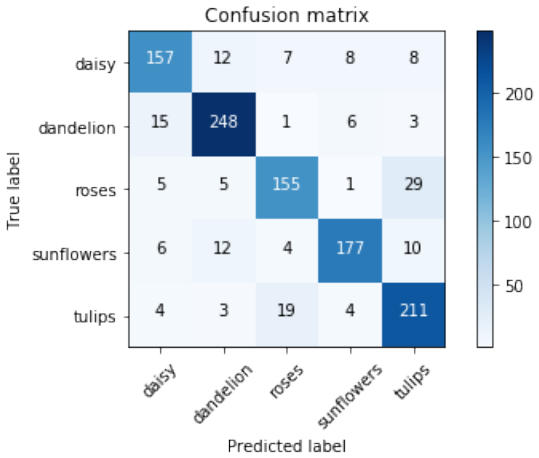


Transfer Learning Example:

Experimental Results:





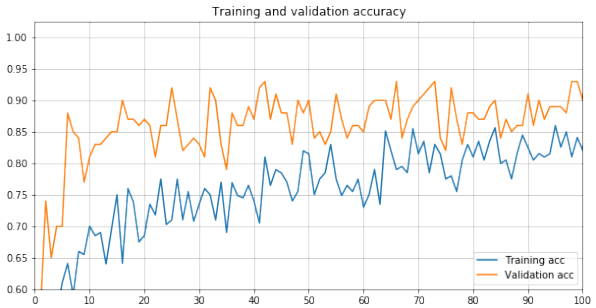


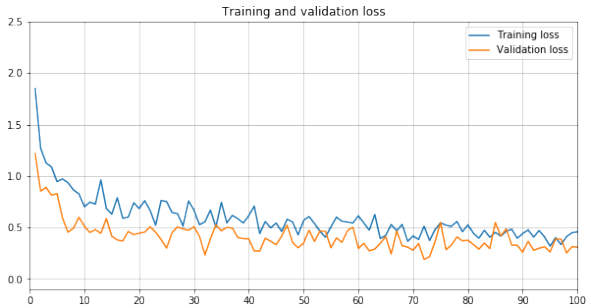
- The discrepancies between the accuracy and loss (error) curves of the training and validation sets indicate that the model may have started to over-fit the training data.
- The confusion matrix shows errors between classes with similar looking flowers, such daisies and dandelions or roses and tulips.

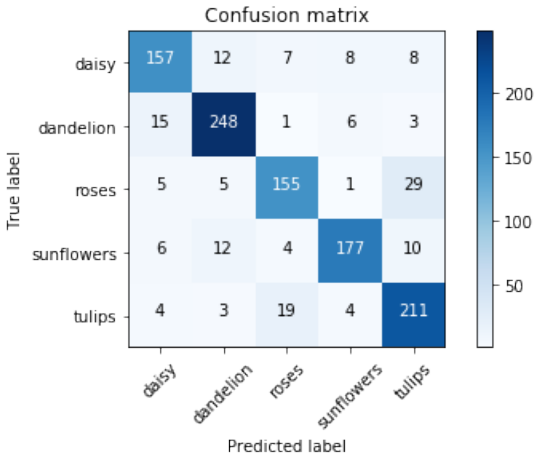
  Some errors (total of 12): Daisy→Dandelion:

Transfer Learning Example:

Experimental Results:

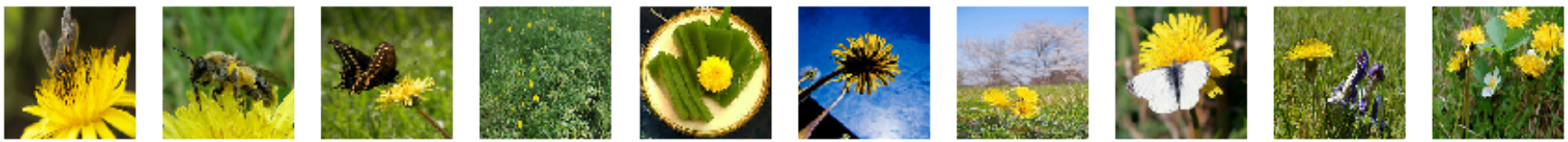






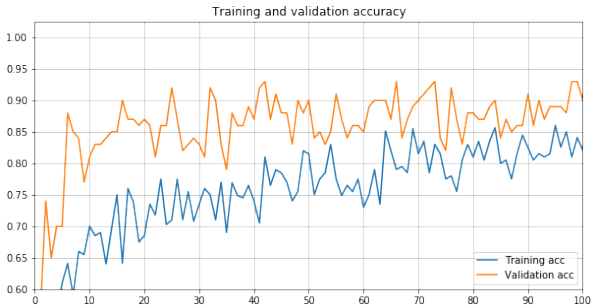
- The discrepancies between the accuracy and loss (error) curves of the training and validation sets indicate that the model may have started to over-fit the training data.
- The confusion matrix shows errors between classes with similar looking flowers, such daisies and dandelions or roses and tulips.

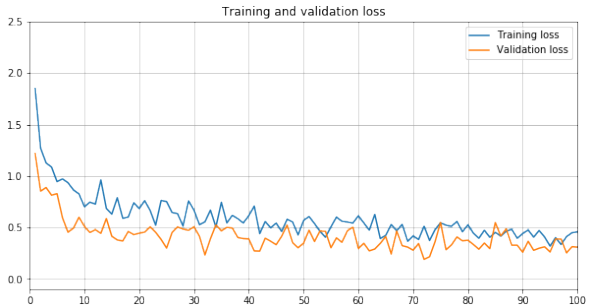
  Some errors (total of 15): Dandelion→ Daisy:

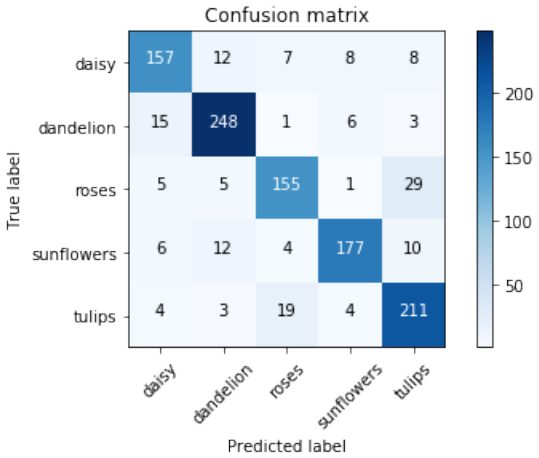


Transfer Learning Example:

Experimental Results:





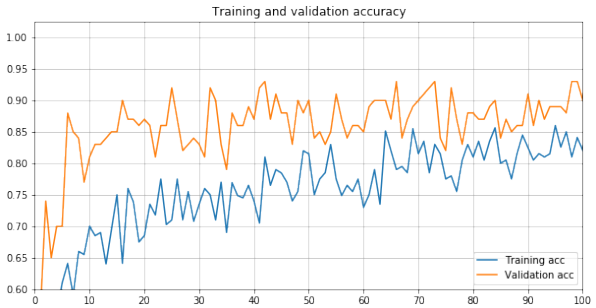


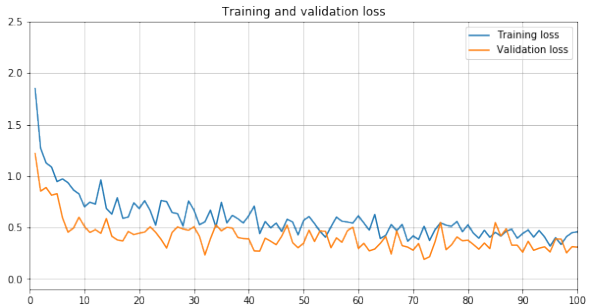
- The discrepancies between the accuracy and loss (error) curves of the training and validation sets indicate that the model may have started to over-fit the training data.
- The confusion matrix shows errors between classes with similar looking flowers, such daisies and dandelions or roses and tulips.

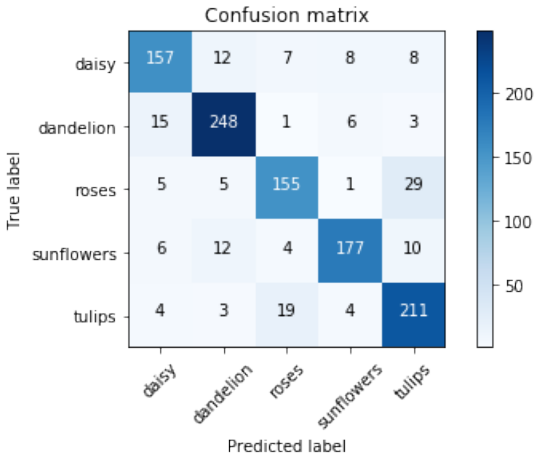
  Some errors (total of 29): Roses→Tulips:

Transfer Learning Example:

Experimental Results:





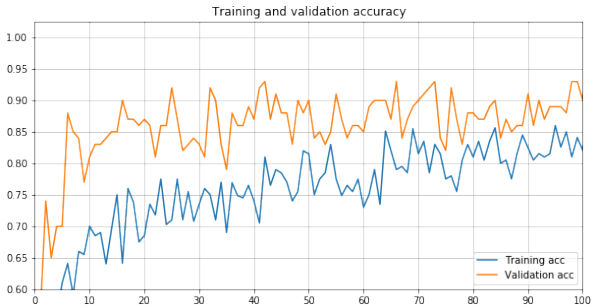


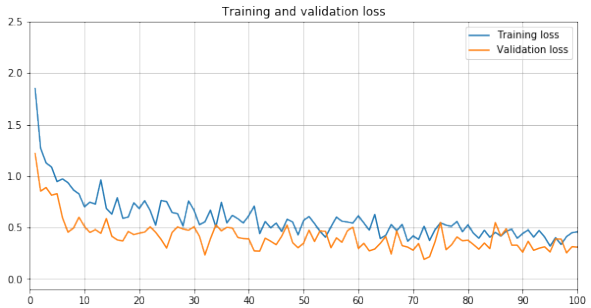
- The discrepancies between the accuracy and loss (error) curves of the training and validation sets indicate that the model may have started to over-fit the training data.
- The confusion matrix shows errors between classes with similar looking flowers, such daisies and dandelions or roses and tulips.

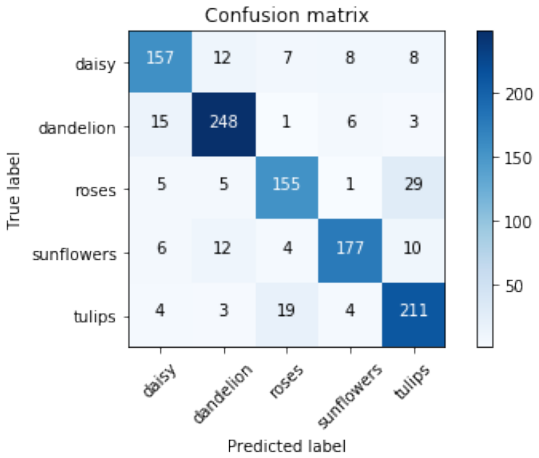
  Some errors (total of 19): Tulips→Roses:

Transfer Learning Example:

Experimental Results:







- To avert over-fitting, we could try increasing the regularization terms in the network weights, or we could used a smaller number of units in the first fully-connected layer. Nevertheless, these are only minor fixes.

The real problem lies in the fact that there is not enough data to adequately train the network!!!

The solution is to use **Data Augmentation**.

Data Augmentation

Data augmentation consists in increasing the number of examples in the training set by transforming the original ones. The resulting images should be as realistic as possible. This way, the model sees many different versions of the original examples, and is exposed to more aspects of the data. This helps generalization.

In Keras, data augmentation can be done by configuring a series of random transformations on the training images in the `ImageDataGenerator`. For example, you can shift, rotate, resize every training image by various amounts. This forces the model to be more tolerant to variations in position, zoom, and orientation of the objects in the images.

```
ImgGen=ImageDataGenerator(rotation_range=30,
                          width_shift_range=0.2,
                          height_shift_range=0.2,
                          shear_range=0.2,
                          zoom_range=0.2,
                          horizontal_flip=True,
                          fill_mode="nearest")
```

`rotation_range`	maximum random rotation in degrees
`with_shift,height_shift`	maximum random shift (as a fraction of the total)
`height_shear`	maximum random shear
`height_zoom`	maximum random zoom
`horizontal_flip`	flip images horizontally
`fill_mode`	interpolation type

Data Augmentation

Data augmentation consists in increasing the number of examples in the training set by transforming the original ones. The resulting images should be as realistic as possible. This way, the model sees many different versions of the original examples, and is exposed to more aspects of the data. This helps generalization.

In Keras, data augmentation can be done by configuring a series of random transformations on the training images in the `ImageDataGenerator`. For example, you can shift, rotate, resize every training image by various amounts. This forces the model to be more tolerant to variations in position, zoom, and orientation of the objects in the images.

Let us take a look at some augmented examples:

```
from keras.preprocessing import image
#select an image
fN="daisy/1150395827_6f94a5c6e4_n.jpg"
#import and convert to numpy array
I=image.load_img(fN,target_size=(200,200))
I=image.img_to_array(I)
#add extra dim
I=I[np.newaxis,:,:,:]
for A in ImgGen.flow(I,batch_size=1):
    A=A.squeeze() #remove extra dim
    plt.imshow(np.uint8(A))
```

Data Augmentation

Data augmentation consists in increasing the number of examples in the training set by transforming the original ones. The resulting images should be as realistic as possible. This way, the model sees many different versions of the original examples, and is exposed to more aspects of the data. This helps generalization.

In Keras, data augmentation can be done by configuring a series of random transformations on the training images in the `ImageDataGenerator`. For example, you can shift, rotate, resize every training image by various amounts. This forces the model to be more tolerant to variations in position, zoom, and orientation of the objects in the images.

Original

Augmented Images

Transfer Learning Example (cont.)

Using Transfer Learning:

- The only change one needs to do is to re-define the generator for the training data.
- **Do not augment the validation and test data**. This will distort the results.

Preparing the Data

```
dbDir="./flowers/"
dataGen1=ImageDataGenerator(preprocessing_function=preprocess_input,
rotation_range=30,width_shift_range=0.2,height_shift_range=0.2,shear_range=0.2,
zoom_range=0.2, horizontal_flip=True,fill_mode='nearest')
dataGen2=ImageDataGenerator(preprocessing_function=preprocess_input)
trainGen=dataGen1.flow_from_directory(directory=dbDir+"train/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
validGen=dataGen2.flow_from_directory(directory=dbDir+"validation/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
testGen=dataGen2.flow_from_directory(directory=dbDir+"test/",
        target_size=(200,200),class_mode="categorical",
        batch_size=32)
```

Two generators, `dataGen1` and `dataGen2` were instantiated using the `ImageDataGenerator` class. `dataGen1` does data augmentation while `dataGen2` does not. `dataGen1` is used only in the training set. `dataGen2` is used to generate the validation and test images.

Transfer Learning Example (cont.)

Create a New Network

```
from keras.applications.vgg16 import VGG16
cnnBaseNew=VGG16(weights="imagenet",include_top=False,input_shape=(200,200,3))
cnnNew=keras.models.Sequential()
cnnNew.add(cnnBaseNew)
cnnNew.add(keras.layers.AvergePooling2D(pool_size=(2,2),strides=2))
cnnNew.add(keras.layers.Flatten()
cnnNew.add(keras.layers.Dense(128,activation="relu"))
cnnNew.add(keras.layers.Dropout(.4))
cnnNew.add(keras.layers.Dense(5, activation="softmax"))
```

Compile and Train

```
opt = keras.optimizers.Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.99)
cnnNew.compile(loss="categorical_crossentropy", optimizer=opt,metrics=["accuracy"])
cnnNew.fit(trainGen,steps_per_epoch=20, validation_data=validGen,
validation_steps=10,epochs=250)
```

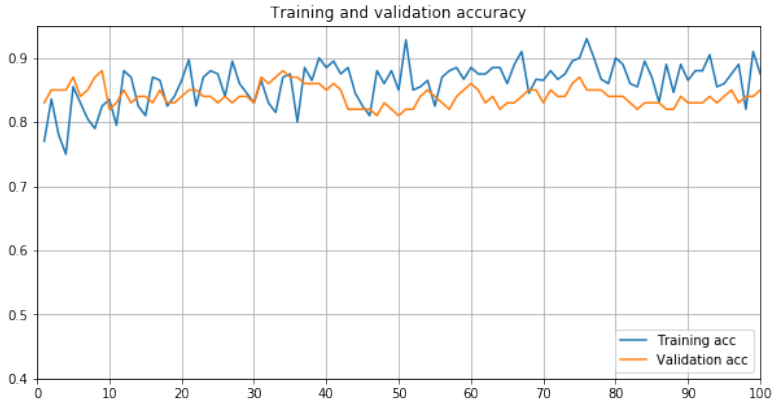
. . . or re-use the previously trained one (cnn)

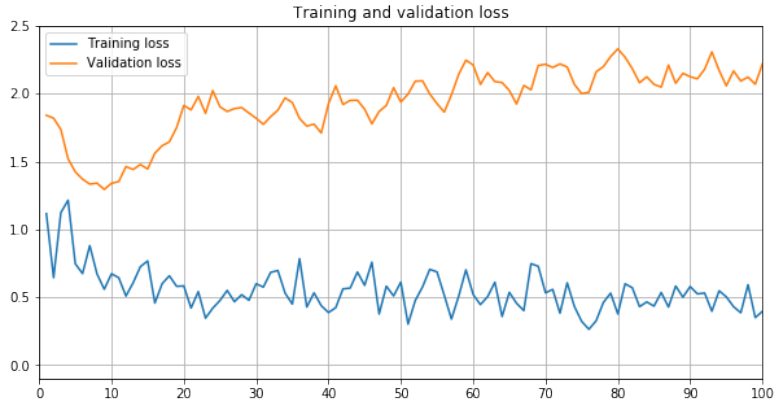
```
opt = keras.optimizers.Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.99)
cnn.compile(loss="categorical_crossentropy", optimizer=opt,metrics=["accuracy"])
cnn.fit(trainGen,steps_per_epoch=20, validation_data=validGen, validation_steps=10,epochs=100)
```

Transfer Learning Example (cont.)

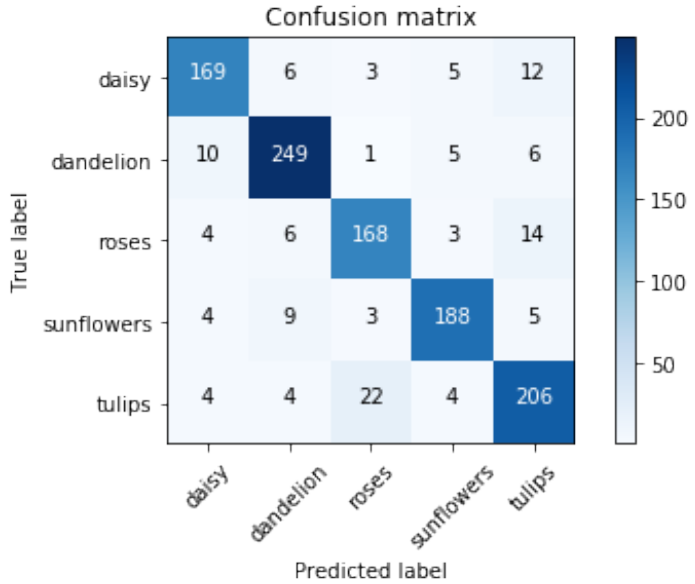
Experimental Results (cnn):

- Accuracies and losses during training for the training and validation sets:





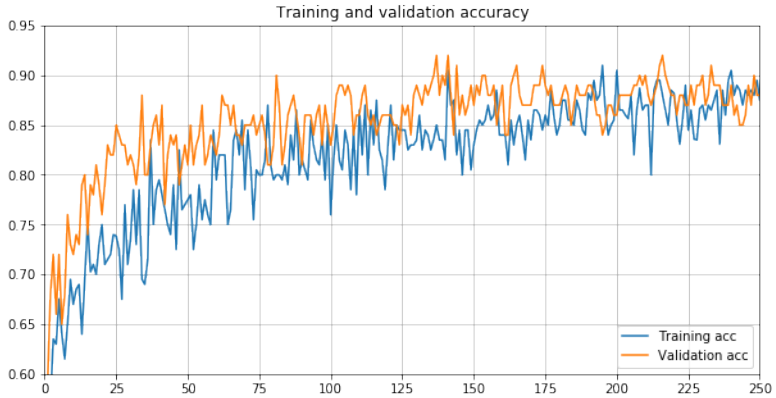
- Test set confusion matrix (accuracy 88.29% - 130 error in 1100 test examples):

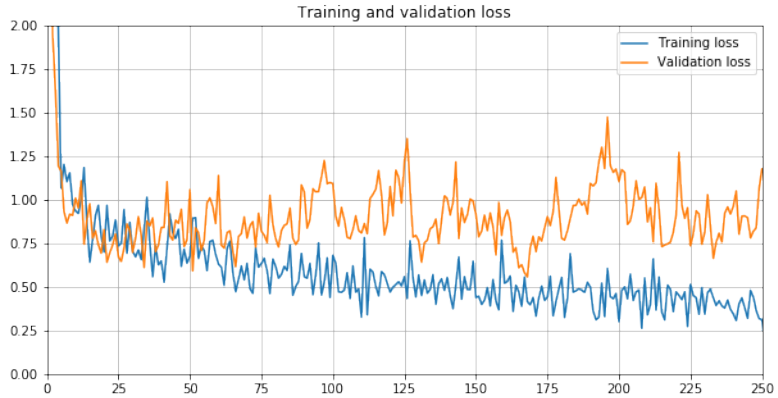


Transfer Learning Example (cont.)

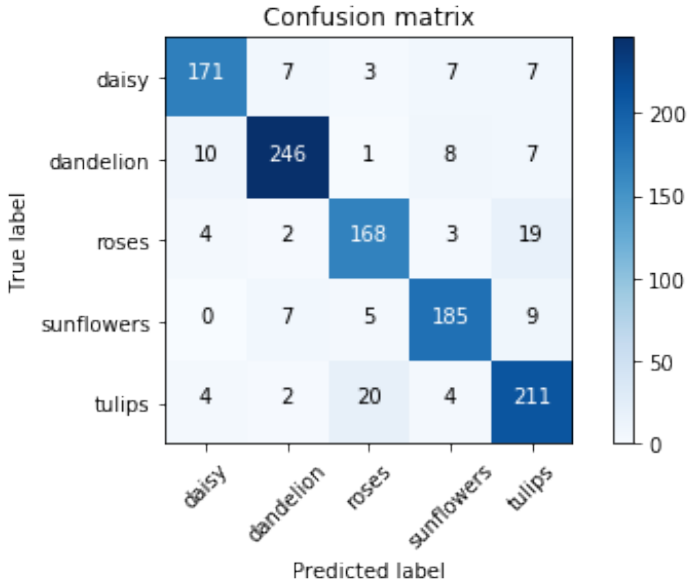
Experimental Results (`cnnNew`):

- Accuracies and losses during training for the training and validation sets:





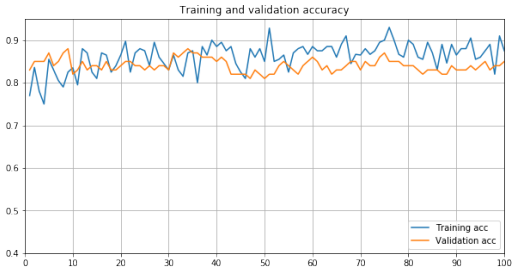
- Test set confusion matrix (accuracy 88.38% - 129 error in 1100 test examples):

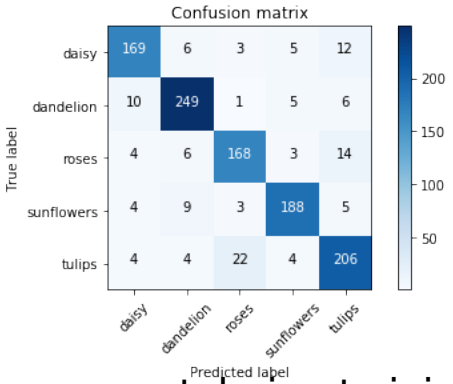


Transfer Learning Example (cont.)

Experimental Results:

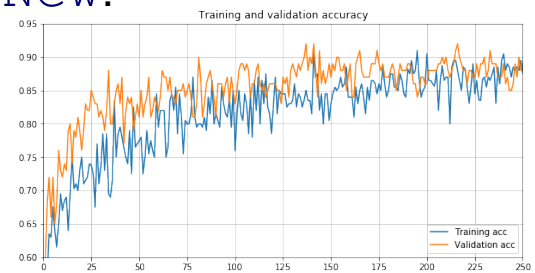
`cnn`:

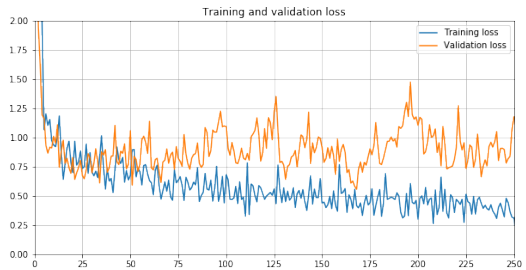


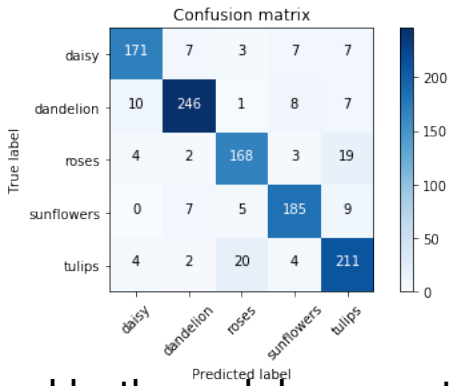


Test set accuracy increased to 88.29%, but there was little improvement during training. Furthermore, the divergence between the loss curves in the training and validation sets indicates some model over-fitting (one should consider re-training with early stopping).

`cnnNew`:







This model's performance is comparable to the previous one, and both models seem to have reached the top of its performances.

The next step is to **fine-tune** the convolutional base layers.

Transfer Learning Example - Fine Tuning

Fine-tuning consist in unfreezing some of the top layer of the convolutional base, and train them along with the newly added MLP network. This will adjust the abstract representations captured in the top convolutional layers to the problem at hand. The step for fine-tuning are the following:

1. Choose a pre-trained network for the convolutional base, and freeze its weights.
2. Add a custom MLP network to the top of the convolutional base.
3. Train the MLP network.
4. Unfreeze some top layer in the convolutional base.
5. Jointly train these layers with the MLP network.

During the training process with data augmentation, our previous models reached an accuracy close to 90% and stop making progress. This means that the MLP network is now trained, and we can unfreeze some of the top layers of the convolutional base. In our case, the convolutional base is a VGG16 network. In this example, only the top convolutional layers will be fine-tuned. To train the whole model is too computationally demanding for most laptop and desktop PCs (unless you have a GPU).

Transfer Learning Example - Fine Tuning

- As previously seen, the top part of the convolutional base is:

```
cnnBase.summary()
Model: "vgg16"
.
.
.
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 25, 25, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 25, 25, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 25, 25, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 12, 12, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 6, 6, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

Transfer Learning Example - Fine Tuning

- Next, the top three layers (`block5_conv`'s) are set to be trainable:

```
layerList=["block5_conv1","block5_conv2","block5_conv3"]
for layer in cnnBase.layers:
    for layerName in layerList:
      if layer.name == layerName:
         layer.trainable=True
```

- Don't forget to compile the model or the changes won't take effect. Set the learning rate to a low value otherwise large updates large may harm the learned representations.

```
opt = keras.optimizers.Nadam(learning_rate=1e-6, beta_1=0.9,beta_2=0.99)
cnn.compile(loss="categorical_crossentropy", optimizer=opt,metrics=["accuracy"])
.
.
.
```

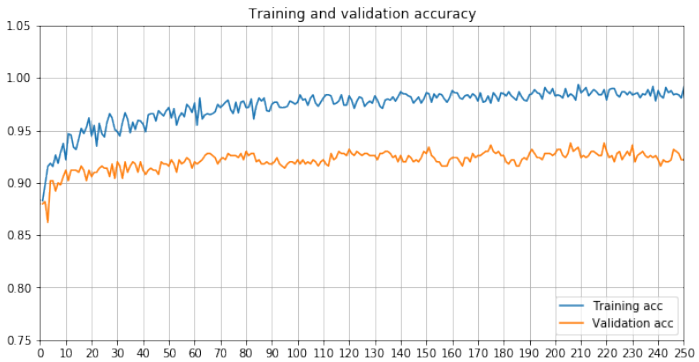
```
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 12, 12, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 12, 12, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 6, 6, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 7,079,424
Non-trainable params: 7,635,26
_________________________________________________________________
```

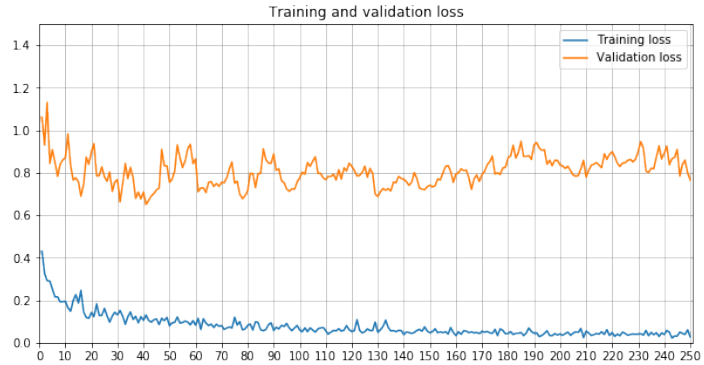
Transfer Learning Example - Fine Tuning

- Now all we need to do is to train the model:

```
cnn.fit(trainGen,steps_per_epoch=20, validation_data=validGen,
        validation_steps=10,epochs=250)
```

- Accuracies and losses during training for the training and validation sets:





- Test set confusion matrix (accuracy 92.61% - 82 error in 1100 test examples):