{EPITECH.}
THE FUTURE OF SOFTWARE
THE BEST OF INNOVATION

Zoidberg2.0

AI & Big Data

Simon MONIER - Mathieu BOULAY - Damien JEANROY
Ayoub BAYED - Maxime SORIN

Software installation

Introduction

This class is technically oriented. A successful student needs to be able to compile and execute Python code that makes use of TensorFlow for deep learning. There are two options for you to accomplish this.

About Python and TensorFlow…

It is possible to install and run Python/TensorFlow entirely from your computer. Google provides TensorFlow for Windows, Mac, and Linux. Previously, TensorFlow did not support Windows. However, as of December 2016, TensorFlow supports Windows for both CPU and GPU operation.

The first step is to install Python 3.9. This is the latest version of Python 3. I recommend using the Miniconda (Anaconda) release of Python, as it already includes many of the data science related packages that are needed by this class.

Anaconda directly supports Windows, Mac, and Linux. Miniconda is the minimal set of features from the extensive Anaconda Python distribution. Download Miniconda from the following URL:

https://docs.conda.io/en/latest/miniconda.html

Use Conda to create a new environment

With conda, you can create, export, list, remove, and update environments that have different versions of Python and/or packages installed in them. Switching or moving between environments is called activating the environment.

```
conda create -y --name tensorflow python=3.9
```

To enter this environment, you must use the following command (for Windows), this command must be done every time you open a new Anaconda/Miniconda terminal window:

```
activate tensorflow
```

And for MacOS:

```
source activate tensorflow
```

Install Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows data scientists to create and share documents that integrate live code, equations, computational output, visualizations, and other multimedia resources.

In addition to jupyter, we will install Scipy which is a Python library used for scientific and technical computing.

```
conda install -y jupyter scipy
```

You should also link your new TensorFlow environment to Jupyter so that you can choose it as a Kernel. Always make sure to run your Jupyter notebooks from your 3.9 kernel.

```
python -m ipykernel install --user --name tensorflow --display-name "Python
3.6 (tensorflow)"
```

And finally, start Jypter:

```
jupyter notebook
```

How do neural networks work?

About Keras

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result as fast as possible is key to doing good research.

Keras is:

- Simple -- but not simplistic. Keras reduces developer cognitive load to free you to focus on the parts of the problem that really matter.
- Flexible -- Keras adopts the principle of progressive disclosure of complexity: simple workflows should be quick and easy, while arbitrarily advanced workflows should be possible via a clear path that builds upon what you've already learned.
- Powerful -- Keras provides industry-strength performance and scalability: it is used by organizations and companies including NASA, YouTube, or Waymo.

About TensorFlow

TensorFlow 2 is an end-to-end, open-source machine learning platform. You can think of it as an infrastructure layer for differentiable programming.

It combines four key abilities:

- Efficiently executing low-level tensor operations on CPU, GPU, or TPU.
- Computing the gradient of arbitrary differentiable expressions.
- Scaling computation to many devices, such as clusters of hundreds of GPUs.
- Exporting programs ("graphs") to external runtimes such as servers, browsers, mobile and embedded devices.

Keras is the high-level API of TensorFlow 2: an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

Keras empowers engineers and researchers to take full advantage of the scalability and cross-platform capabilities of TensorFlow 2: you can run Keras on TPU or on large clusters of GPUs, and you can export your Keras models to run in the browser or on a mobile device.

First contact with Keras

The core data structures of Keras are layers and models. The simplest type of model is the Sequential model, a linear stack of layers. For more complex architectures, you should use the Keras functional API, which allows you to build arbitrary graphs of layers, or write models entirely from scratch via subclassing.

Here is the Sequential model:

```
from tensorflow.keras.models import Sequential

model = Sequential()
```

Stacking layers is as easy as `.add()`:

```
from tensorflow.keras.layers import Dense

model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

Once your model looks good, configure its learning process with `.compile()`:

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

If you need to, you can further configure your optimizer. The Keras philosophy is to keep simple things simple, while allowing the user to be fully in control when they need to (the ultimate control being the easy extensibility of the source code via subclassing).

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(learning_rate=0.01,
momentum=0.9, nesterov=True))
```

You can now iterate on your training data in batches:

```
# x_train and y_train are Numpy arrays
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Evaluate your test loss and metrics in one line:

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Or generate predictions on new data:

```
classes = model.predict(x_test, batch_size=128)
```

What you just saw is the most elementary way to use Keras.

However, Keras is also a highly-flexible framework suitable to iterate on state-of-the-art research ideas. Keras follows the principle of **progressive disclosure of complexity**: it makes it easy to get started, yet it makes it possible to handle arbitrarily advanced use cases, only requiring incremental learning at each step.

Models

Starting from scratch

First, we created a neural network model from scratch, referring to the Keras documentation:

```
classifier = Sequential()

classifier.add(Conv2D(32, (3, 3), input_shape =(img_dims,img_dims2,3),
activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

classifier.add(Conv2D(64, (3, 3), input_shape = (img_dims,img_dims2,3),
activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

classifier.add(Conv2D(128, (3, 3), input_shape = (img_dims,img_dims2,3),
activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

classifier.add(Conv2D(256, (3, 3), input_shape = (img_dims,img_dims2,3),
activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

classifier.add(Flatten())
classifier.add(Dense(units = 128, activation = 'relu'))
classifier.add(Dense(units = 64, activation = 'relu'))
classifier.add(Dense(units = 32, activation = 'relu'))
classifier.add(Dense(units = 1, activation = 'sigmoid'))
```

Then, we compiled it by defining the metrics, here "accuracy":

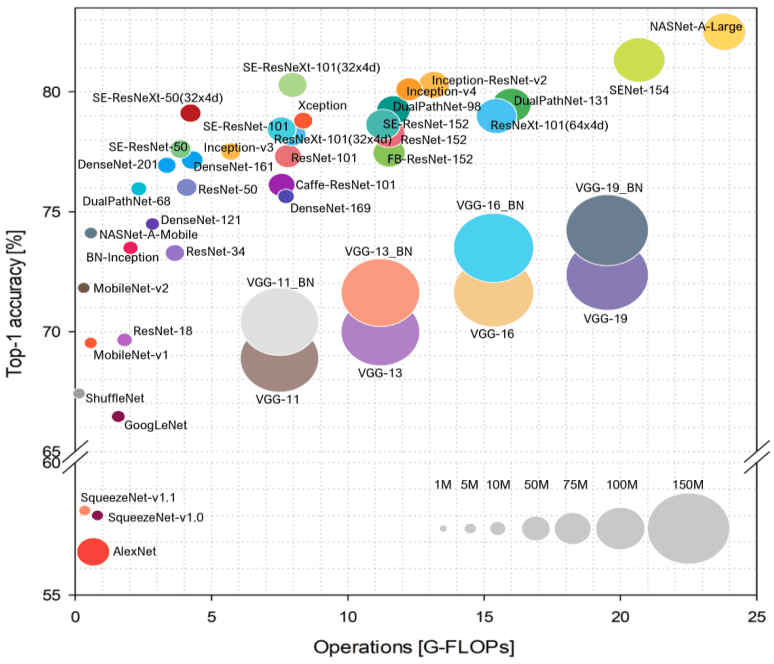
```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
= ['accuracy'])
```

Then we defined a training stop in case of stagnation during the training of the model:

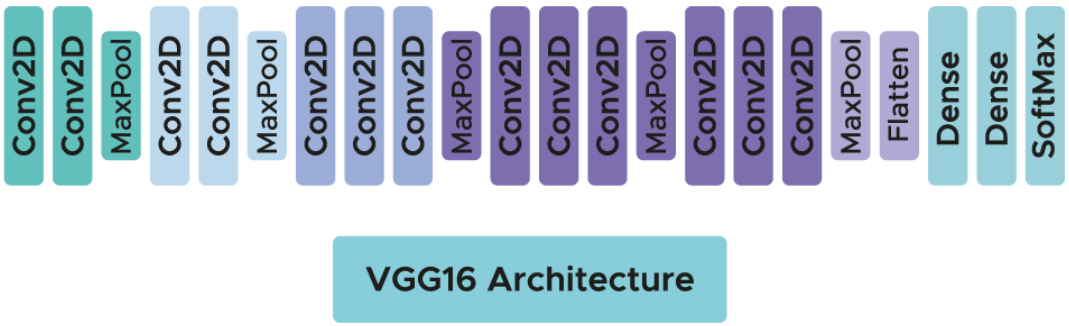
```
epochs = 20
early_stop = EarlyStopping(monitor='val_loss', patience=2)

Enfin, nous avons entraîné notre premier modèle :

hist = classifier.fit_generator(training_set,
steps_per_epoch=training_set.samples // batch_size,
                                epochs=epochs, validation_data=test_set,
                                validation_steps= test_set.samples,
callbacks=[early_stop])
```

The first models were not very efficient, because we were not sure how many layers of neurons we should put and how many neurons per layer. Moreover, we lacked metrics to better appreciate the results obtained.

After several tests with different neural network sizes, we were able to gain a good understanding of Keras and we decided to implement the best model architecture currently available.



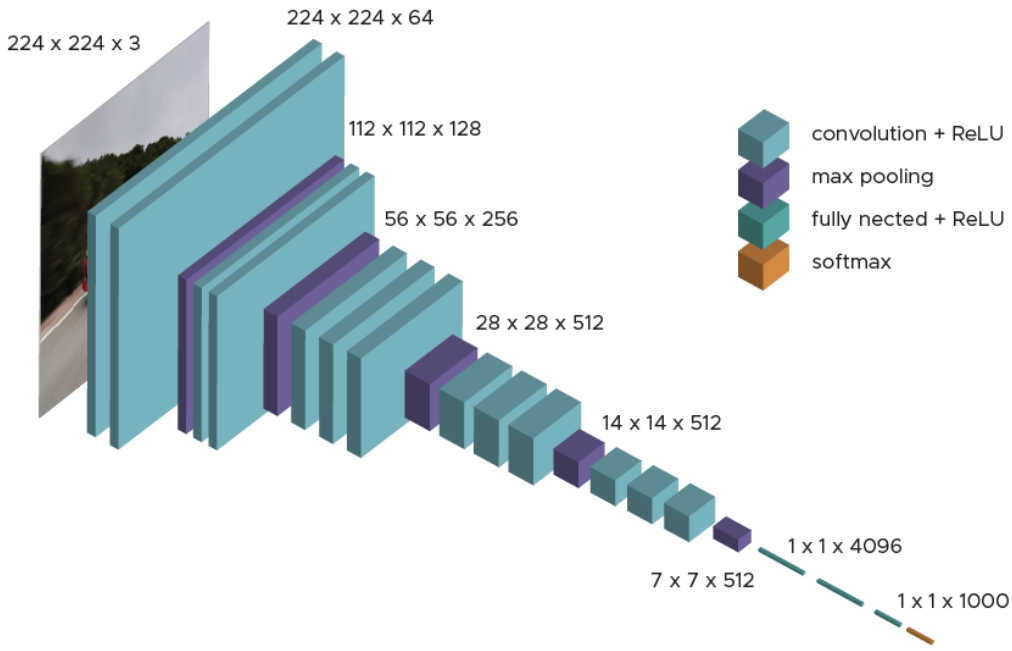
Implementation of the VGG16 model



VGG is a convolutional neural network proposed by K. Simonyan and A. Zisserman from Oxford University and gained notoriety by winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014.

The model achieved an accuracy of 92.7% on Imagenet which is one of the highest scores achieved. It marked an improvement over previous models by proposing convolution kernels in the convolution layers with smaller dimensions (3×3) than what had been done before.

In fact there are two algorithms available: VGG16 and VGG19. Here we will focus on the architecture of the first one. If the two architectures are very close and follow the same logic, VGG19 presents a larger number of convolution layers.



During model training, the input for the first convolution layer is an RGB image of size 224 x 224. For all convolution layers, the convolution kernel is of size 3×3: the smallest dimension to capture the notions of top, bottom, left/right and center. This was a specificity of the model at the time of its publication.

Until VGG16 many models were oriented towards convolution kernels of larger dimension (size 11 or size 5 for example). Recall that the purpose of these layers is to filter the image by keeping only discriminating information such as atypical geometric shapes. These convolution layers are accompanied by Max-Pooling layers, each of size 2×2, to reduce the size of the filters during the learning process.

At the output of the convolution and pooling layers, we have 3 layers of Fully-Connected neurons. The first two are composed of 4096 neurons and the last one of 1000 neurons with a softmax activation function to determine the image class.

As you can see the architecture is clear and simple to understand which is also a strength of this model. We have modified the model so that it only classifies 3 categories. This implies that we had to re-train the model.

The implementation of the model is simple and fast, a few lines of code are enough:

```
from keras.applications.vgg16 import VGG16
Model = VGG16(include_top=True, weights=None, input_shape=input_shape)
```

We can visualize the layers of the model with this command:

```
Model.summary()

Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
```

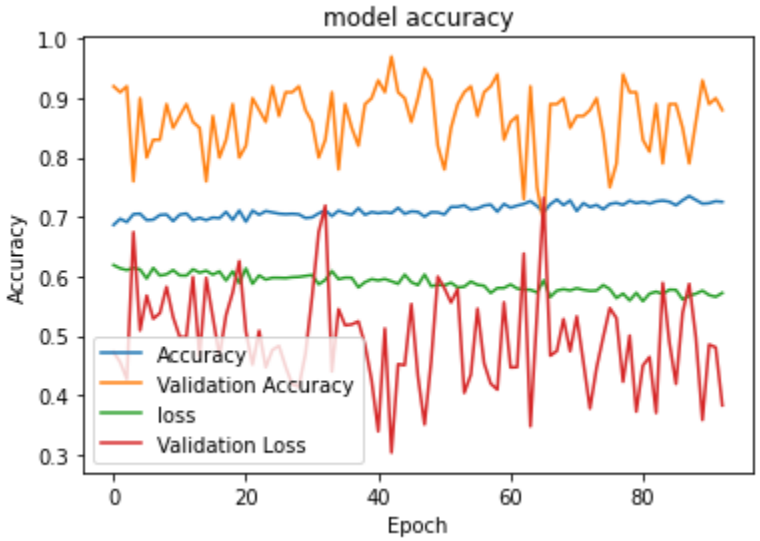
```
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 4096)              102764544
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dense_2 (Dense)              (None, 8192)              33562624
_________________________________________________________________
dense_3 (Dense)              (None, 16384)             134234112
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 49155
=================================================================
Total params: 302,106,435
Trainable params: 287,391,747
Non-trainable params: 14,714,688
```

We have added to the training stop in case of stagnation, a checkpoint (backup) of the model, to save the best training parameters:

```
from keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint = ModelCheckpoint(filepath="vgg16_1.h5", monitor =
'val_accuracy',verbose=1, save_best_only=True)

Early=EarlyStopping(monitor='val_accuracy',min_delta=0,patience=50,verbose=1
, mode='auto')
```

Once the model was trained, we evaluated it:

```
score = vgg_model.evaluate(test_set)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

216/216 [==============================] - 12s 52ms/step - loss: 0.4249 -
accuracy: 0.8830
Test loss: 0.42487770318984985
Test accuracy: 0.8829540014266968
```



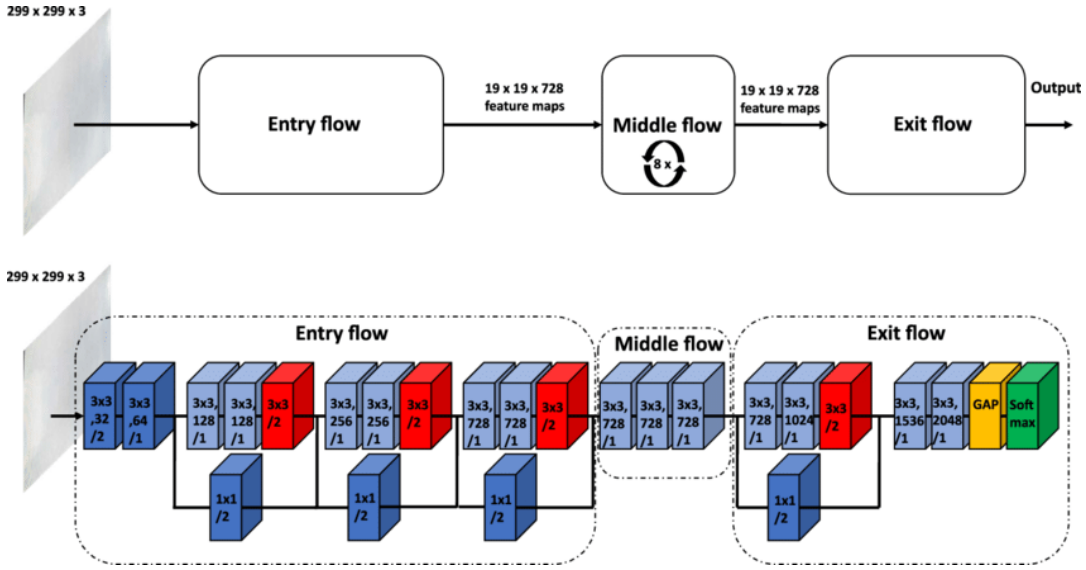
We had 88% accuracy, which was a very good start. However, we had a lot of false negatives, which would be a problem if we went into production. We talk about the improvement we made on this issue later in this report.

Implementation of the Xception model

The Xception model is proposed by François Chollet. Xception is an extension of the Inception architecture that replaces standard Inception modules with depth-separable convolutions.

Xception stands for "extreme inception" and takes the principles of Inception to the extreme. In Inception, 1x1 convolutions were used to compress the original input, and from each of these input spaces, we used different types of filters on each of the depth spaces. Xception simply reverses this step.

Instead, it first applies the filters to each of the depth maps, and then finally compresses the input space using 1X1 convolution by applying it to the entire depth. This method is nearly identical to depth-separable convolution, an operation that has been used in neural network design as far back as 2014. There is another difference between Inception and Xception.

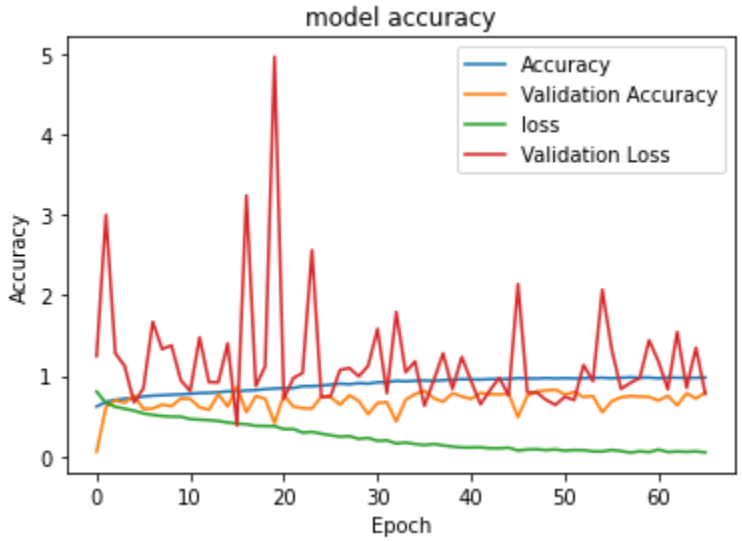


We modified the model so that it only classifies 3 categories. This implies that we had to re-train the model. The implementation of the model is simple and fast, only a few lines of code are needed thanks to Keras :

```
from keras.applications.xception import Xception
Model =
Xception(classes=n_classes,classifier_activation="softmax",weights=None,
input_shape=input_shape)
```

Once the model was trained, we evaluated it:

```
186/186 [==============================] - 14s 52ms/step - loss: 0.7034 -
accuracy: 0.8525
Test loss: 0.7034318447113037
Test accuracy: 0.8525295853614807
```

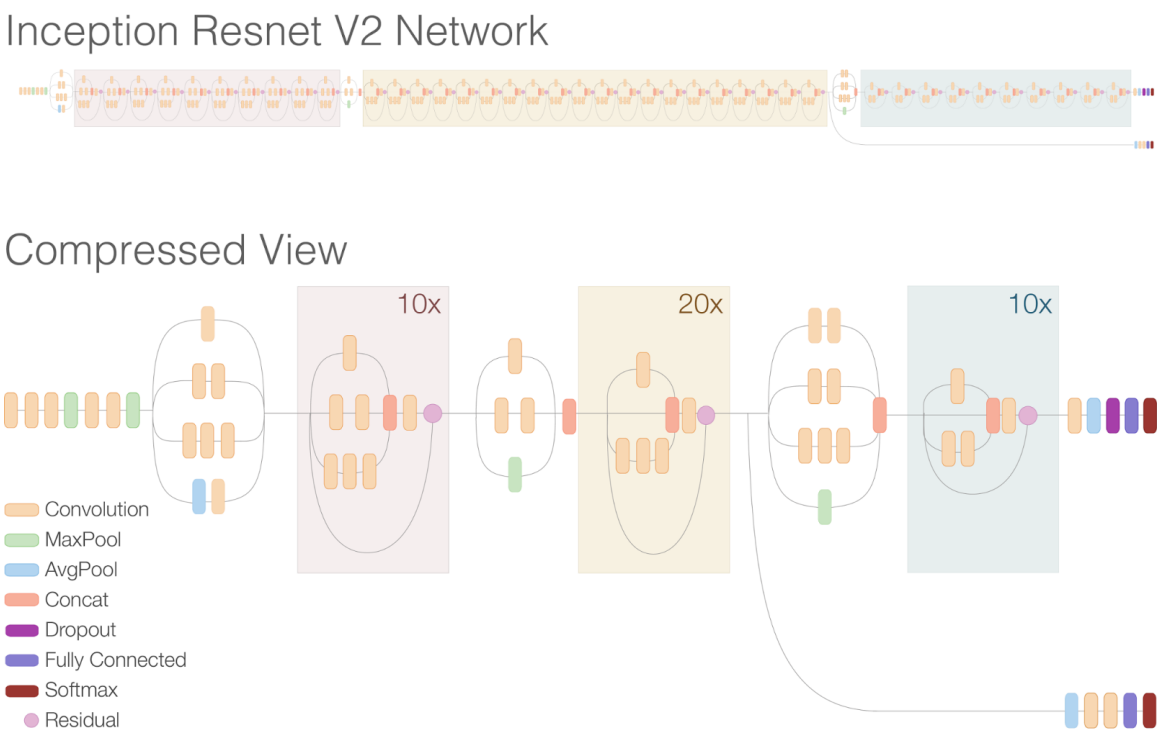


We had 85% accuracy, the model was a little less accurate than VGG16. However, we had a lot of false negatives, which would be a problem if we went into production. We talk about the improvement we made on this, later in this report.

Implementation of the InceptionResNetV2 model

ResNet and Inception have been at the heart of the most significant advances in image recognition performance in recent years, with very good performance at relatively low computational cost. Inception-ResNet combines the Inception architecture with residual connections.

Inception-ResNet-v2 is a convolutional neural network. The network has a depth of 164 layers and can classify images into 1000 object categories, such as keyboard, mouse, pencil and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 299 by 299.



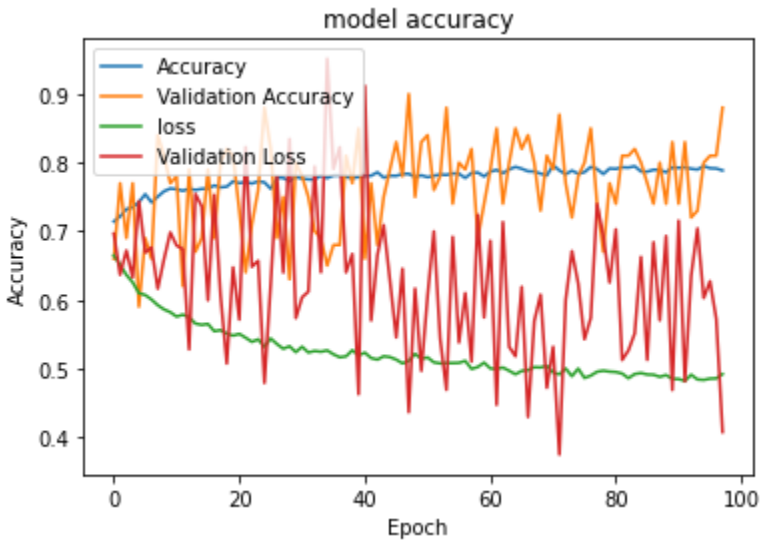
We modified the model so that it only classifies 3 categories. This implies that we had to re-train the model.

The implementation of the model is simple and fast, only a few lines of code are needed thanks to Keras :

```
from tensorflow.keras.applications.inception_resnet_v2 import
InceptionResNetV2
conv_base = InceptionResNetV2(include_top=False, weights=None,
input_shape=input_shape)
```

Once the model was trained, we evaluated it.

```
63/63 [==============================] - 8s 116ms/step - loss: 0.5920 -
accuracy: 0.8253
Test loss: 0.5919711589813232
Test accuracy: 0.8253205418586731
```



We had an accuracy of 82%, so the model was a little less accurate than Xception.
However, we had a lot of false negatives, which would be a problem if we went into production. We talk about the improvement we made on this issue later in this report.

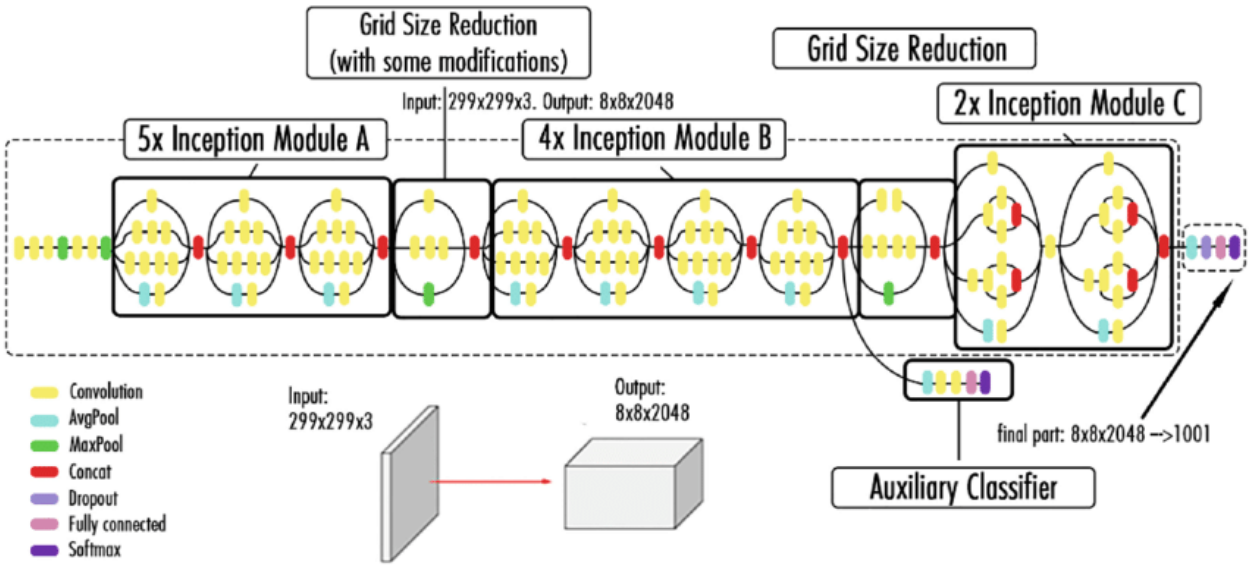
Implementation of the InceptionV3 model

Inception V3 is a deep learning model based on convolutional neural networks, which is used for image classification. Inception V3 is a superior version of the basic Inception V1 model that was introduced as GoogLeNet in 2014. As the name suggests, it was developed by a team at Google.

What makes the Inception V3 model better?

Inception V3 is just the advanced and optimized version of the Inception V1 model. The Inception V3 model used several network optimization techniques for better model adaptation. It has a higher efficiency. It has a deeper network compared to the Inception V1 and V2 models, but its speed is not compromised. It is less computationally expensive. It uses auxiliary classifiers as regularization.

The inception v3 model was released in 2015, it has a total of 42 layers which is a bit higher than the previous inception V1 and V2 models and a lower error rate than its predecessors. The efficiency of this model is really impressive.

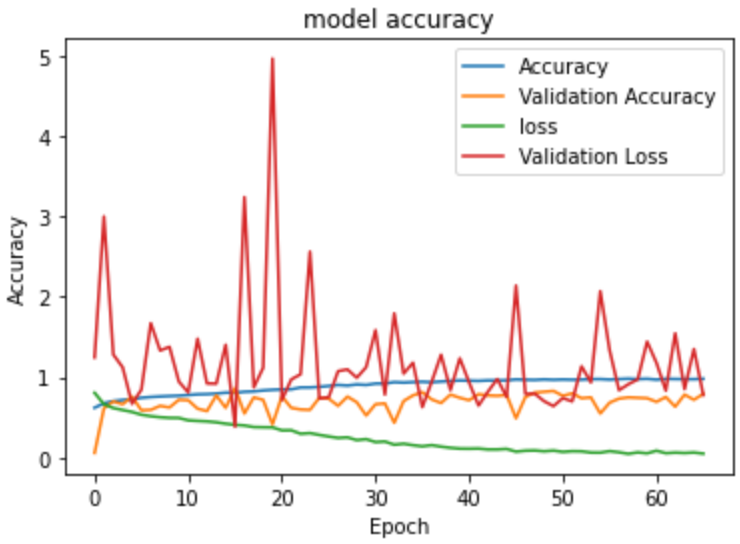


We modified the model so that it only classifies 3 categories. This implies that we had to re-train the model. The implementation of the model is simple and fast, only a few lines of code are needed thanks to Keras :

```
from tensorflow.keras.applications.inception_v3 import InceptionV3
Model =InceptionV3(classes=n_classes,classifier_activation="softmax",
                   weights=None,input_shape=input_shape)
```

Once the model was trained, we evaluated it:

```
93/93 [==============================] - 9s 99ms/step - loss: 0.5629 -
accuracy: 0.8229
Test loss: 0.5629357099533081
Test accuracy: 0.8229278922080994
```



We had an accuracy of 82%. However, we had a lot of false negatives, which would be a problem if we went into production. We talk about the improvement we made on this issue later in this report.

Implementation of the NasNet model

We tried to implement the NASnet model which is the best model currently in terms of accuracy, but we lack hardware resources to run the model. We could not implement this model which requires a lot of memory and computing power.

Implementation of the Random Forest

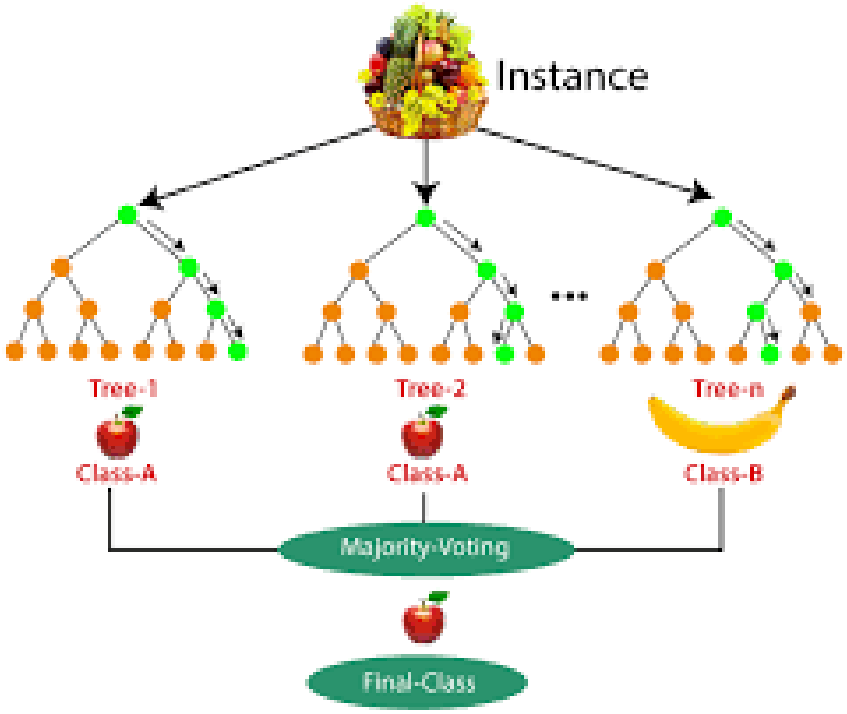
We then tried other forms of artificial intelligence, other than neural networks. The Random Forest algorithm is a classification algorithm that reduces the variance of the predictions of a single decision tree, thus improving their performance. To do this, it combines many decision trees in a bagging approach.

The random forest algorithm was proposed by Leo Breiman and Adèle Cutler in 2001. In its most classical form, it performs parallel learning on multiple randomly constructed decision trees trained on different subsets of data.

The ideal number of trees, which can go up to several hundred or more, is an important parameter: it is very variable and depends on the problem. Concretely, each tree of the random forest is trained on a random subset of data according to the bagging principle, with a random subset of features (variable characteristics of the data) according to the "random projections" principle. The predictions are then averaged when the data are quantitative or used for a vote for qualitative data, in the case of classification trees.

The random forest algorithm is known to be one of the most efficient "out-of-the-box" classifiers (i.e. requiring little data preprocessing).

It has been used in many applications, including consumer ones, such as for the classification of images from the Kinect* game console camera in order to identify body positions.



To implement this algorithm, we used Scikit-learn. Scikit-learn is a free Python library for machine learning. It is developed by many contributors, especially in the academic world by French higher education and research institutes such as Inria. It offers in its framework many libraries of algorithms to implement, ready to use. These libraries are available to data scientists in particular.

It includes functions for estimating random forests, logistic regressions, classification algorithms, and support vector machines. It is designed to harmonize with other free Python libraries, notably NumPy and SciPy.

The implementation of this algorithm was more complicated to realize, a description of each step of its implementation can be found in the notebook "random_forest". The presentation here would be too long, so we refer you to the notebook. However, after training this algorithm, we obtained an accuracy of 74%. Much less efficient than the neural networks we implemented before.

Improved prediction

Dataset balancing

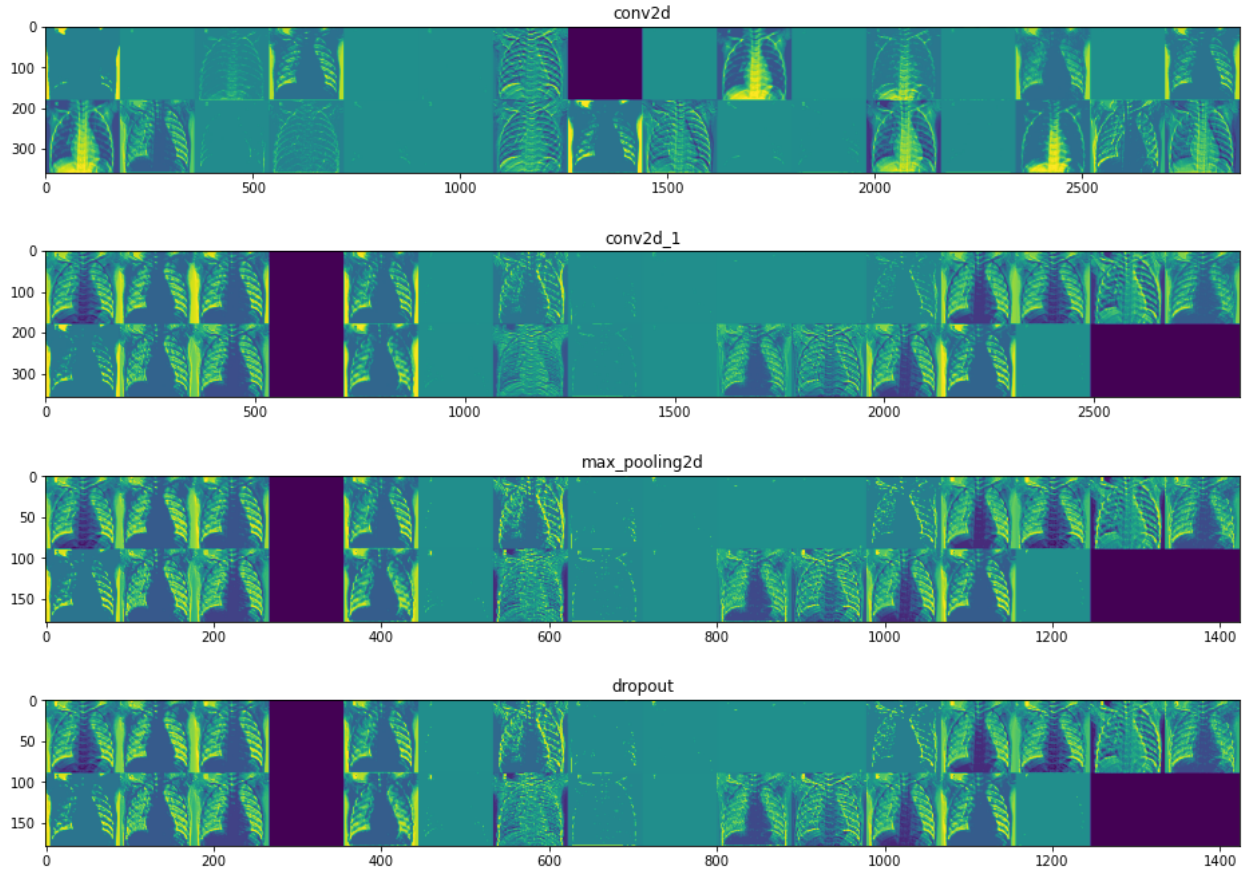
We balanced the dataset to improve learning and accuracy. We have about 1300 images of bacterial pneumonia and 1300 of viral pneumonia. We reduced the number of normal lung images to 1000, so we hope to get more false positives rather than false negatives. For a physician, a false positive is better than a false negative in the case of diagnostic assistance.

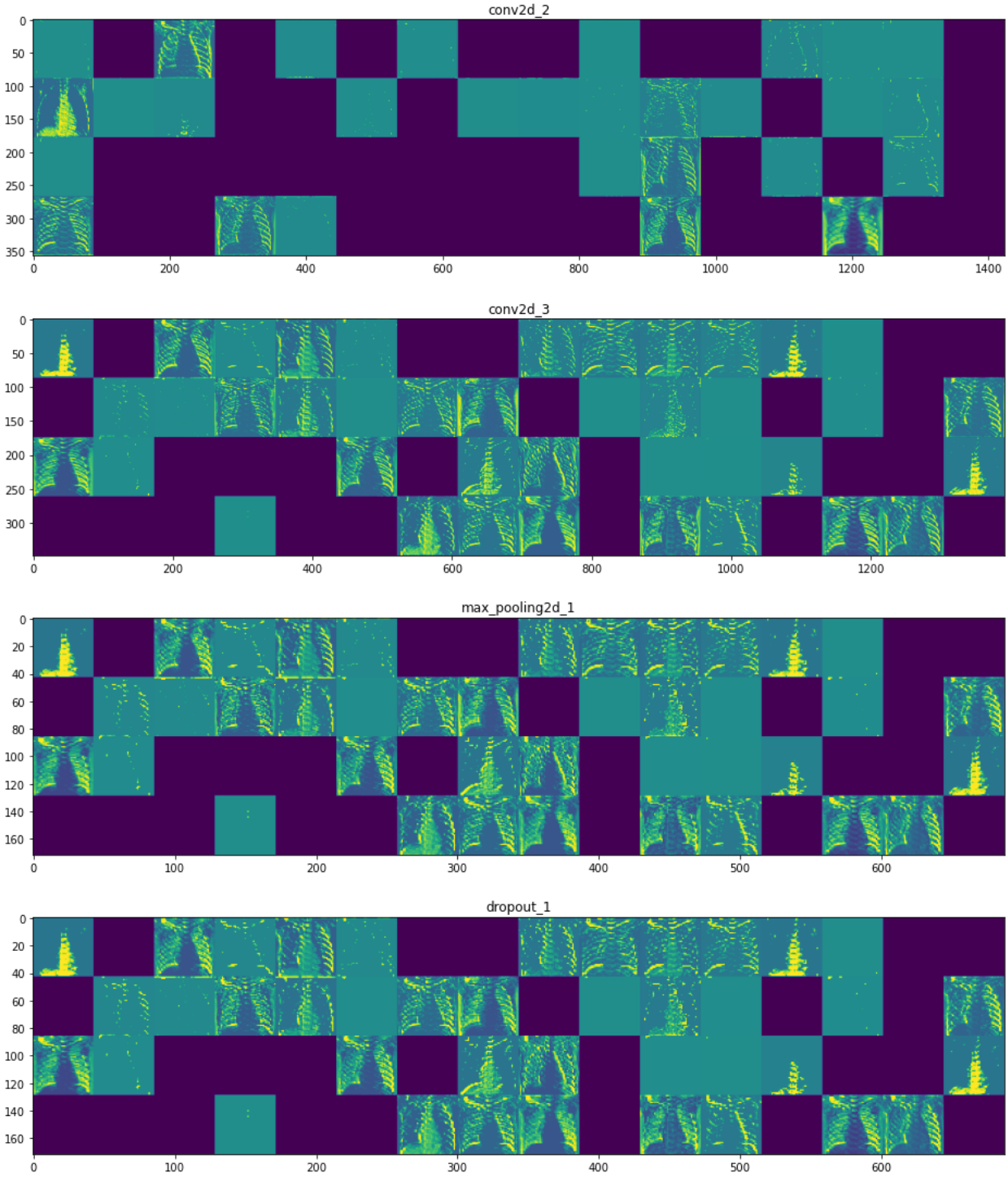
Visualization of the convulsion layers

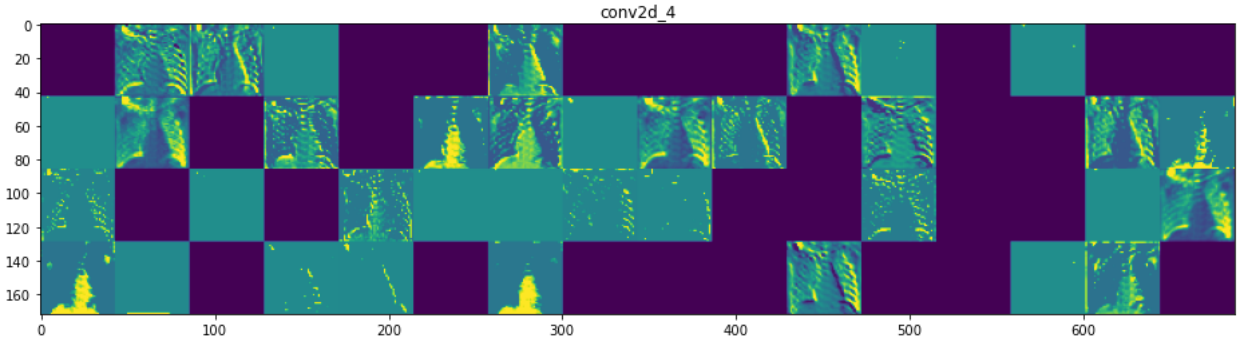
We have reproduced some of the work of François Chollet in his book Deep Learning with Python in order to learn how our layer structure processes the data in terms of visualizing each intermediate activation.

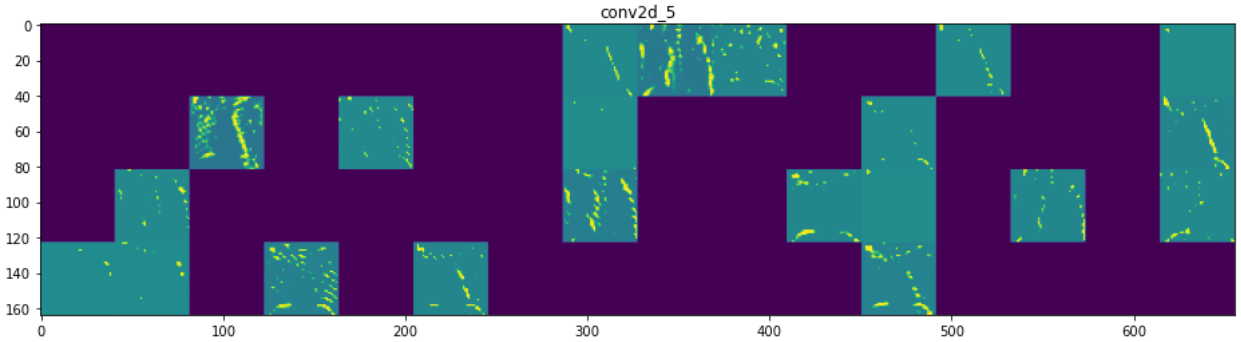
This consists of displaying the feature maps that are produced by the convolution and pooling layers in the network. What this means is that we have visualized the result of each activation layer. To see all the details of this implementation, we invite you to consult the notebook "visualization-neuron-network-convolution".

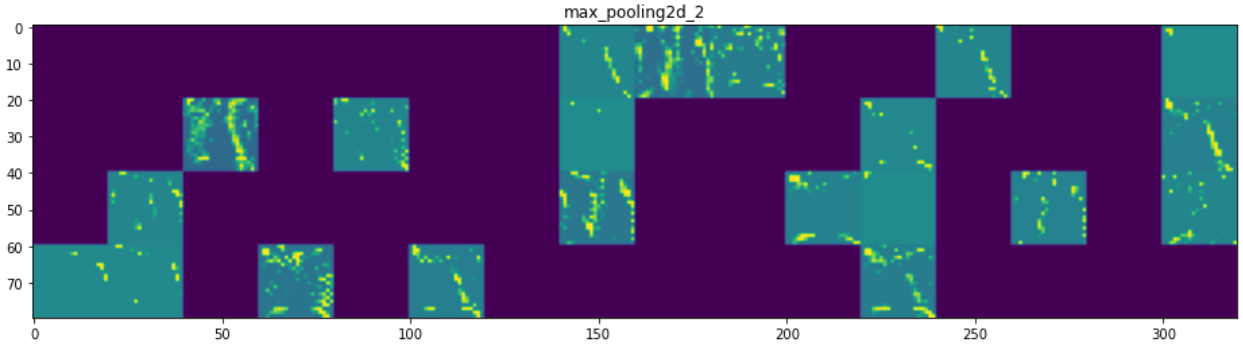
Here is what we have obtained:

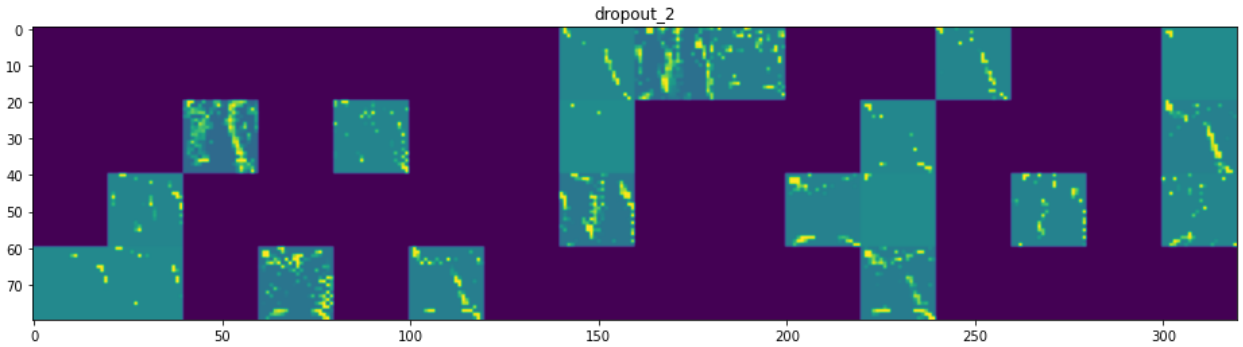



conv2d_2
conv2d_3
max_pooling2d_1
dropout_1


conv2d_4
0
20
40
60
80
100
120
140
160
0
100
200
300
400
500
600


conv2d_5
0
20
40
60
80
100
120
140
160
0
100
200
300
400
500
600


max_pooling2d_2
0
10
20
30
40
50
60
70
0
50
100
150
200
250
300


dropout_2
0
10
20
30
40
50
60
70
0
50
100
150
200
250
300

We can say that the first layer retains the complete shape of the radio, although several filters are not activated and are left in green. At this point, the activations retain almost all the information present in the initial image.

As we move down the layers, the activations become more abstract and less visually interpretable. They begin to encode higher level concepts such as simple edges, corners and angles. The higher presentations convey less and less information about the visual content of the image, and more and more information related to the class of the image.

As mentioned above, the structure of the model is overly complex, to the point where we can see our last layers not activating at all, there is nothing left to learn at this point. So we visualized how a convolutional neural network finds patterns in some basic figures and how it carries information from one layer to another.

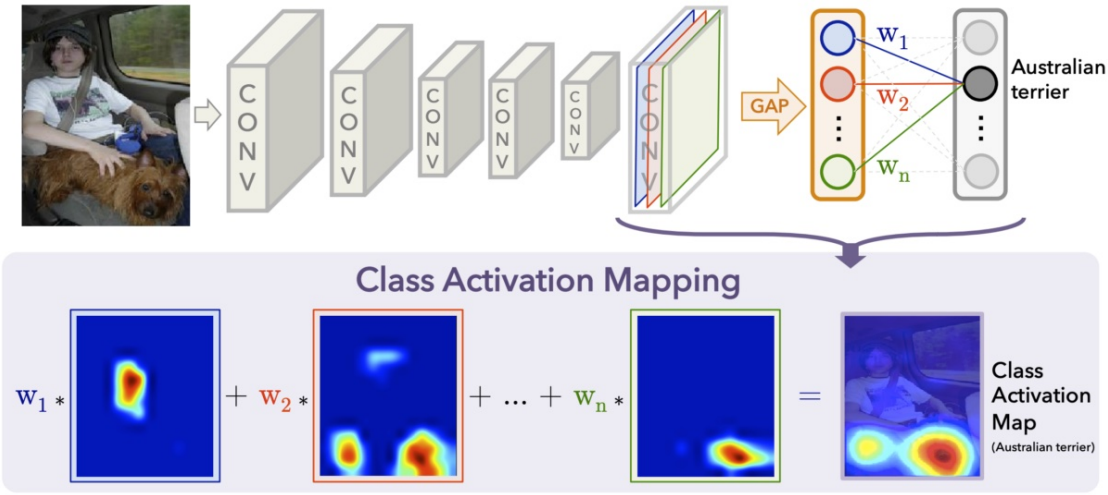
Set up gram-cam

Grad-CAM consists in searching which parts of the image led a convolutional neural network to its final decision. This method consists in producing heat maps representing the activation classes on the images received as input. An activation class is associated with a specific output class.

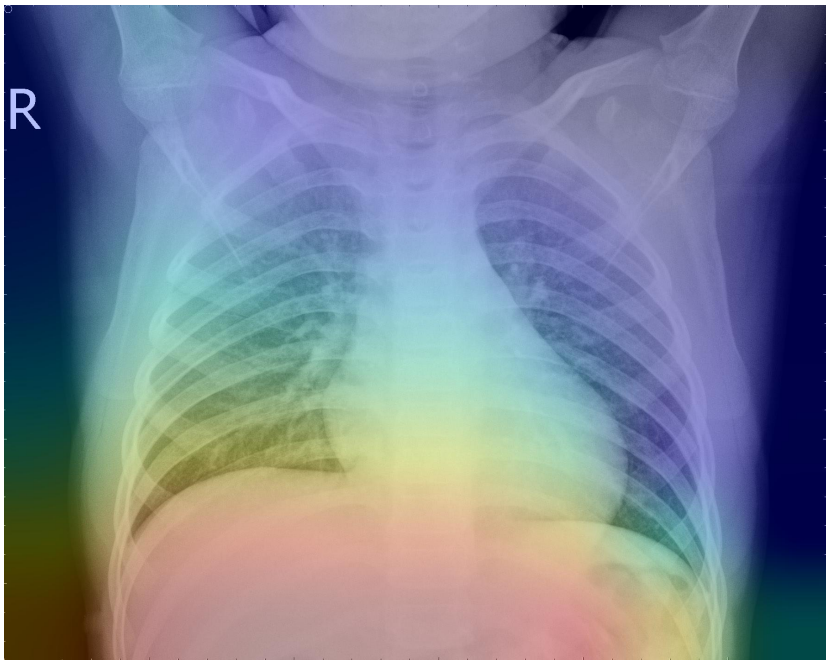
These classes will indicate the importance of each pixel in relation to the class concerned by increasing or decreasing the intensity of the pixel. For example, if an image is used in a convolutional network of cats and dogs, the Grad-CAM visualization will generate a heatmap for the "cat" class, indicating to what extent the different parts of the image correspond to a cat, and also a heatmap for the "dog" class, indicating to what extent the parts of the image correspond to a dog.

For example, let's consider a CNN of dogs and cats. The Grad-CAM method will generate a heatmap for the cat object class to indicate to what extent each part of the image corresponds to a cat, and also a heatmap for the dog object class by proceeding in the same way.

The class activation map assigns importance to each position (x, y) in the last convolutional layer by computing the linear combination of activations, weighted by the corresponding output weights for the observed class (Australian terrier in the example below). The resulting class activation mapping is then resampled to the size of the input image. This is illustrated by the heatmap below.



To see its implementation, we invite you to consult the "Grad-cam" notebook. Here are the results we were able to obtain:



We have used this new technique to interpret convolutional neural networks which are a state-of-the-art architecture, especially for image-related tasks. Research in the area of interpretable machine learning is progressing at an accelerated pace and is proving to be very important in gaining user confidence and contributing to model improvement.

This allows us to see if our neural network relies on the right part of the radio to make its prediction.

Set up gradient-centralization

We implemented Gradient Centralization, a new optimization technique for deep neural networks by Yong et al. Gradient Centralization can both speed up the training process and improve the final generalization performance of DNNs. It operates directly on the gradients by centralizing the gradient vectors to have a zero mean. Gradient Centralization also improves the Lipschitzness of the loss function and its gradient so that the learning process becomes more efficient and stable.

Optimization techniques are of great importance to efficiently and effectively train a deep neural network (DNN). It has been shown that using first- and second-order statistics to perform Z-score normalization on network activations or weight vectors, such as batch normalization (BN) and weight normalization (WS), can improve the training. performance.

Different from these existing methods that operate primarily on activations or weights, we present a new optimization technique, namely gradient centralization (GC), which operates directly on gradients by centralizing gradient vectors to have a zero mean. GC can be viewed as a projected gradient descent method with a constrained loss function. It is shown that GC can regularize both the weight space and the output feature space to improve the generalization performance of DNNs. Moreover, GC improves the Lipschitzness of the loss function and its gradient so that the training process becomes more efficient and stable.

We invite you to consult the Gradient-centralization notebook to see its implementation and the results obtained.

Comparison of accuracy between models