Chapter 9 : Part 5

**Deep Learning**

**CNN: Image Recognition Project**

**9.5.1 Problem Description**

A *CNN* is just an *ANN* on which you use *convolution* *trick* to add some *Convolutional* *layers*. We use this *Convolutional* *layers* to preserve the *special* *structure* in *images* So that we can *classify* some *images*.

* CNN are great deep learning models for computer vision, to classify some images/photographs, or even some videos.
* Problem Description: In this section, we are not going to solve any business problem as we used to do in the previous sections. Here we are simply going to solve an *image* *classification* *problem*.
* We will have some images of *cats* and *dogs* and we will train a *CNN* to predict if the *image* is a photo of a *dog* or of a *cat*.
* Where we'll have a *folder* full of *images*, these ***images will be some*** images of *cats* and *dogs*.

|  |  |
| --- | --- |
| * Classify any image: Once we build our *CNN* *model* you will simply need to *change* the images of *cats* and *dogs* in the folder and replace them by the *images* you want to work with. * For example, you will be able to *replace* these *cats* and *dogs* images by some *medical* *images* such as: *brain* *image* contains a *tumor* or not. If you know the answers of ***enough observations*** (like ***thousands*** of ***observations***), then you will be able to train a CNN to predict if some new *brain* *image* contains a *tumor* or not. * CNN can use to accelerate cancer research as we can see in this article right here. |  |

**9.5.2 Work environment for CNN**

Work environment for CNN will be different than other projects that are done in previous chapters. Because remember, the data-set we used to work with were tables, containing *some* *independent* variables and *one* *dependent* variable.

* Now we have some images, so we need to do some image pre-processing to be able to input these images in our CNN.
* We have a folder named "*dataset*" which contains *10,000 images* of *cats* and *dogs*. These images are *pre-categorized* and named as "*cat.0001.jpeg*" or "*dog.0067.jpeg*" in *jpeg-image-format*.
* This ***"dataset"*** folder must be a *sub-folder* of our *working* *directory* where we have ***CNN.py*** file.

**9.5.3 Data preprocessing : folder structure**

Previously we had *.csv* file but here the independent variables are now the *pixels* distributed in *3D arrays*, and therefore we *cannot* add *explicitly* the *dependent* *variable* in our *dataset* because it wouldn't make much sense to add this dependent variable column along the *3D* arrays *representing* the images.

* Remember, when we *train* a *ML* model we always need the *dependent* *variable* to have the *real* *results* that are required to *understand* the *correlations* between the *information*.
* But here, *since* we cannot add this *dependent* *variable* *column* in the same table, how can we extract the info of this dependent variable?
* We have several solutions:

1. *Categorize* each *image* by giving *category* & *Number* to its *image* *file* name. Eg: "*cat.0001.jpeg*" or "*dog.0067.jpeg*". Then split all files into train-test.

* A classic solution is to only have a dataset containing our images, separated in two different folders, *training\_set* and *test\_set*.
* Name each of these images by the category, for example, as "*cat.0001.jpeg*" or "*dog.0067.jpeg*" a number to differentiate all the images. In each folder the *training* *set* and the *test* *set*, we would get, for example, *5,000* images of *cats* and *5,000* images of *dogs*.
* Then we can *write* some kind of *code* to *extract* the *label* name *Cat* or *Dog* from the *name* of the *image* *file* to specify to the *algorithm* whether this *image* belongs to the *class-Cat* or belongs to the *class-Dog*.
* And in some way, we get the our *dependent* variable *vector*, by *filling* this *dependent* *variable vector* with the *label* *names* (cat/dog) that we managed to *extract* from the *image file names* of all our images.

1. *Categorize* each *image* by creating *different* *folder*, eg: "**cats**", "**dogs**" for each train & test folder. We're gonna use that in **Keras**.

* Other solution, comes with Keras, it contains some *tricks* and *tools* to *import* some *images* in a very efficient way. And that's the solution we'll use.
* Folder Structure for Keras: To import the images with *Keras*, we only need a *special* *folder* *structure* for our dataset.
* To split training set and a test set, we create two sub-folders inside ***"dataset"*** folder named "***test\_set***" and "***training\_set***". Inside each of those 2 folders we create two more folders named "***cats***" and "***dogs***".
* Each of "***cats***" and "***dogs***" inside "***training\_set***" have **4000** cats-images and **4000** dogs-images respectively.
* Each of "***cats***" and "***dogs***" inside "***test\_set***" have **1000** cats-images and **1000** dogs-images respectively.
* Then for total ***10000*** images we've divided the data into ***0.8*** for *train-data* and ***0.2*** for *test-data*.
* The first pillar of the structure is to separate your images into two separate folders, we already said that, a training set folder and a test set folder.
* But that's *not* the *main* *point* !!! remember we want to have a simple way to differentiate the class labels (i.e. ***cats*** and ***dogs***).
* To differentiate the ***cat images*** and the ***dog images***, the simple trick is to make two different folders named "***cats***" and "***dogs***" one folder for the cats and one folder for the dogs. We have to make "***cats***" and "***dogs***" folders inside of each "***test\_set***" and "***training\_set***" folders.
* But remember it is not essential to name each image as: "*cat.0001.jpeg*" or "*dog.0067.jpeg*", it could be "*0001.jpeg*" or "*0067.jpeg*" because those images are now in separate folders. Images are now categorized by the folder now. (But the files that we got from Kaggle are already named as "***cat.0001.jpeg***" or "***dog.0067.jpeg***").
* And that's how Keras will understand how to differentiate the labels of your dependent variable.
* Those *images* can be *any kind* of *images* you have on your *computer*, can take some pictures of your *friends* and replace these *dog's* pictures (☺ because dogs are also our best friend ☺) by the pictures of your *friends* and then you'll be able to *train* an algorithm that will *predict*, which *friend* of yours is in the *pictures*. So that can be pretty fun to do, but *remember*, you need a *lot* *of images*.
* Where to find this dataset: This dataset is a very well-known dataset in *computer* *vision*, it can be used as a *performance* *benchmark*, to test your *Deep Learning models* on this to simply see if you get some *good* *accuracy* and so it's a very useful dataset.
* Our dataset here is actually a subset of the whole dataset, that you can find on *Kaggle*. Because the original whole dataset contains *25,000* images, but here is just a subset we have *10000* image.
* Size Of Our Dataset: The size of our dataset is same as the dataset of the business problem we had in the ANN. The train-test split will be similar.
* We have *10,000* images in total in the dataset, *8,000* images in the *training* *set* and *2,000* images in the *test* *set*. So that's an 80% 20% split, then in the training set we have *4,000* images of *dogs* and *4,000* images of *cats*. And in the test set we have *1,000* images of *dogs* and *1,000* images of *cats*.
* No encoding needed: We don't need to encode any *categorical* *data* because, of course, our *independent* *variables* are in someway the *pixels* in the three *R-G-B channels*. So, there is no categorical data here and therefore we don't need to do any *encoding*.
* splitting the datasets: We did it already splitted into two folders "***test\_set***" and "***training\_set***".
* Feature scaling: Of course we *need* *feature* *scaling*. Feature scaling is *100% compulsory* in *deep* *learning* and especially for *computer* *vision*.
* Previously *feature* *scaling* section was associated to *data pre-processing* part. But here we are not using any previous pre-processing techniques, so we will take care of *feature scaling later*, just before we fit our *CNN* to our images.
* So some part of *data pre-processing* was done *manually*. We do *feature* *scaling* and image *augmentation*, so that our deep learning *models* can run the most *efficiently* as possible.
* Hence the first part of our *CNN* model *won't be* our usual *data pre-processing*, we built the *CNN* *first*.

**9.5.4 Import packages for CNN**

The first step is to import all the *Keras* packages. Following are the only packages we'll need to make our *convolutional neural network*.

# *=========== ====  Convolutional Neural Network : CNN  ==== ============*

# *----------- Install following  packages -------------*

    # *Install Theano*

    # *Install Tensorflow*

    # *Install Tensorflow*

    # *Install Keras*

# *---------- Part 1 - Building the CNN ----------------*

# *Importing the Keras libraries and packages*

**from** keras.models **import** Sequential     # *to initialize as sequence-of-layers*

**from** keras.layers **import** Convolution2D  # *Convolution step for images*

**from** keras.layers **import** MaxPooling2D   # *not "MaxPool2D". Pooling step for images*

**from** keras.layers **import** Flatten        # *Flattening step*

**from** keras.layers **import** Dense          # *ads fully-connected-layers to classic ANN*

1. The first package is *Sequential*, and we already in *ANN*. we'll use it to *initialize* our *neural network* because, remember, there are *two* *ways* of initializing the neural network, either as a sequence of layers or as a graph. And since a *CNN* is still a *sequence* of *layers*, well, we use the ***Sequential*** package to initialize our neural network.
2. Second package, ***Convolution2D***, is the package is used for the *first* *step* of making the *CNN*, the *Convolution* *Step*, in which we add the *convolutional* layers.
3. Since we're working on *images* and since images are in *two dimensions* (unlike, for example, *videos* that are in *three dimensions* with the *time*) hence, ***Convolution2D*** package to deal with *images*.
4. ***MaxPooling2D***, used to proceed to step two, the *Pooling Step*, that will add our *pooling* *layers*.
5. Next package, ***Flatten*** used for 3rd step, *flattening*, in which we *convert* all the pooled *feature* *maps* (that we created through *Convolution* and *MaxPooling*) into a *large* *feature* *vector* that is then becoming the *input* of our *fully connected layers*.
6. The ***Dense*** package is used it in the *ANN* section. This is the package we use to *add* the *fully* *connected* *layers* (of CNN) in a classic *ANN*.

Basically each of above packages corresponds to one step of the construction of the CNN.

**9.5.5** **Initialize our CNN**

To initialize our CNN, we provoke the *Sequential* package. It's exactly the same as initializing a *classic* ANN. We are going to create an object of the ***Sequential*** class, and we're gonna call this object ***cnn\_classifier***.

* This *classifier* will classify some images, to tell if each image is a picture of a *dog* or a *cat*. So we're doing nothing *else* than *classification*.

# *initializing the CNN*

cnn\_classifier = **Sequential**()

**9.5.6** **Adding Layers: step 1 : Convolution - layer**

|  |  |
| --- | --- |
| Here we add different *layers*. The *first* *layer* that we're going to add is the *convolutional* layer.  A quick reminder of the building process: the *CNN* *building* *process* takes *four* *steps*.   1. Step one, ***Convolution*** 2. step two, ***Max*** ***Pooling***, 3. step three, ***Flattening*** and 4. step four, ***Full*** ***Connection***.   Here we will complete the first step, Convolution. |  |
|  |

* Convolution Step: Here we have an input image of a *cat* or a *dog*, We convert into a *table* of *pixel* *values*. *Convolution* *step* consists of applying *several* *feature* *detectors* on this *input* *image*.

# *step 1 : Convolution - layer*

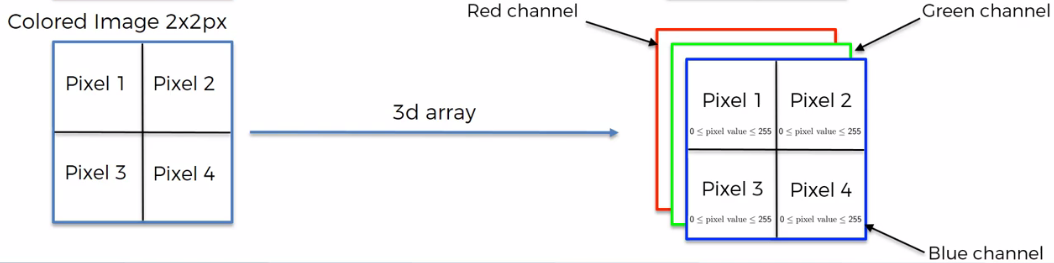
cnn\_classifier**.add**(**Convolution2D**(32, 3, 3, input\_shape = (64, 64, 3), activation= "relu"))

* For example: In our intuition section we studied a *smile* *face* image (above),
* Here the *feature detector* is the feature detector of the left side of a smiling mouth when we *slide it* all over the *input image* and when the feature detector *passes over* the part of the *face* that contains this *left side* of the *mouth* that is *smiling*, we get a *high number* in this table (notice the 4 on ***feature*** ***map***).
* So for each feature detector that we apply on the input image we get a feature map.
* The *feature* *map* contains some *numbers* and the *highest* *numbers* of the *feature* *map* is where the *feature detector* could *detect* a *specific* *feature* in the *input image*.
* Number of feature detectors: We will choose the *number* of *feature detectors*, therefore the *number* of *feature* *maps* in this step.
* We get as many *feature maps* as *feature detectors* we use to detect some *specific* *features* in the *input* *image*. And those *feature* *maps* will form our *Convolution-layer*.
* We are going to apply an ***add()*** method on our ***cnn\_classifier*** object (we used it in *ANN* to create a *classic layer* composed of *several* *nodes*, but here we used it to create a *Convolution-layer*).
* Parameters of our Convolution-layer: Remember when we added the classic layer in the ***ANN***, we used the ***dense()*** function, which is used to add a fully connected layer (hidden-layer) in the ANN and therefore this is not the *function* that we're gonna use here (we'll use it later).
* The function that we're gonna use is ***Convolution2D***,
* Parameters for ***Convolution2D()***:
* ***nb\_filter***: the number of *filters*. It specifies the number of *feature-detectors/filters* that we're going to apply on our input image to get this ***same number*** of *feature* *maps*.

|  |  |
| --- | --- |
| * So the *number* of *filters* that we choose here is the *feature* *maps* that we want to create as well because there will be *one* *feature* *map* created for *each* *filter* used. * row and column size of filter/feature detector: number of feature detectors is not the only thing that we need to choose here, we also need to specify row and column size of filter/feature detector/convolution kernel (convolution kernel is just another name for feature detector or filter).   cnn\_classifier.add(Convolution2D**(32, 3, 3,**  Here we set ***no. of filters*** = **32**, ***row\_size*** = **3**, ***column\_size =* 3*.*** |  |

|  |
| --- |
| * Why we choose number of filters as 32? Because, * Common Practice is 32: Most of the CNN architectures, the common practice is to start with ***32***. Therefore our ***convolutional*** ***layer*** will be composed of ***32*** ***feature*** ***maps***. * If we start with *32* *feature detectors* in the *first* *convolutional* layer and then we can add other *convolutional* *layers* with more feature detectors like *64* and then *128* and then *256*. * We have no GPU: The second reason we are working on a CPU. |

* ***border\_mode*** (optional): Just to specify how the feature detectors will *handle* the *borders* of the *input* *image*. But most of the time we choose keyword- "**same**", as the default value.
* We don't need to input it because default is automatically applied.
* ***input\_shape:*** Very important argument. ***input\_shape*** is the *shape* of your *input* *image* on which you are going to apply your *feature* *detectors* through the convolution operation.
* Since all of our images don't have the same size, same format and therefore we need to force them in some way having the same format.
* Therefore, specifying ***input\_shape*** will *convert* all of our images into one same *single* *format* and therefore *one* *fixed* *size* of the image.
* Remember, we will do this conversion during the Image Pre-Processing part, right after we build our CNN and just before we fit our CNN to our images.



* We know that:
* If the image is a colored image then input images are converted into 3D arrays,
* If the image is a black-and-white image then input images are converted into 2D arrays.
* Since we are working with *colored* *images*, our images will be *converted* into *3D arrays* during the *image pre-processing* part.
* This 3D array is composed of three channels, each channel corresponding to one color, Red, Green or Blue (R-G-B), and each channel corresponds to single 2D array that contains the pixels of our images.

For example, use **input\_shape=(128, 128, 3)** for 128x128 RGB pictures in RGB channel. You can use **None** when a dimension has variable size.

cnn\_classifier.add(Convolution2D(32, 3, 3, **input\_shape = (64, 64, 3)**,

* ***3*** is the ***number*** ***of*** ***channels*** (since we are dealing with the colored image), it will only be **1** if we're dealing with a black and white image.
* ***64*** and ***64*** are the *dimensions* of the *2D array* in *each* of the *channel*.
* All of that means that we are expecting colored images of pixels.

|  |
| --- |
| * Why small size: Here we are using a smaller format because we're using a CPU. This will be way enough to get some good accuracy results if you're working on a GPU, and we don't wanna wait too many hours to execute the code. * You can choose a larger format like *128* by *128* or even *256* by *256,* but either you need to use a *GPU* or run your code before sleeping for *8 hours*. * Why 3 channels (colored): We are keeping *three* *channels* of color information because *cats* and *dogs* don't have the *same* *colors* and therefore *differentiating* them with the *colors* can be helpful to classify them. * Notice the order of the ***input\_shape*** parameters here. * Theano back-end: *Number of channels* first and then the *Dimensions of the 2D arrays*. This order is used in **Theano back-end** (old version ??).   input\_shape=(3, 128, 128)   * TensorFlow back-end: The *Dimensions Of The 2D Arrays* first and then *Number of channels*. This order is used in **TensorFlow back-end**.   **input\_shape=(64, 64, 3)** |

* ***activation:*** We also specify rectifier activation function type in ***Convolution2D***,
* We already used it in *Hidden-layers (fully connected layers)* in the ANN in the previous chapter. In ANN we used **activation= "relu"** to *activate* the *neurons* in the NN.
* But in CNN we used **activation= "relu"** to make sure we get *non-linearity* in all of our *feature maps*:
* It will make sure that we don't have any negative pixel values in our feature maps. Because depending on the parameters that we use for our convolution operation, we can get some negative pixels in the feature map.
* We need to remove these negative pixels in order to have non-linearity in our CNN.
* Because *classifying* some images is a *nonlinear problem* so we need to have *non-linearity* in our *model*.

cnn\_classifier.add(Convolution2D(32, 3, 3, input\_shape = (64, 64, 3), **activation= "relu"**))

So that’s our convolution-layer:

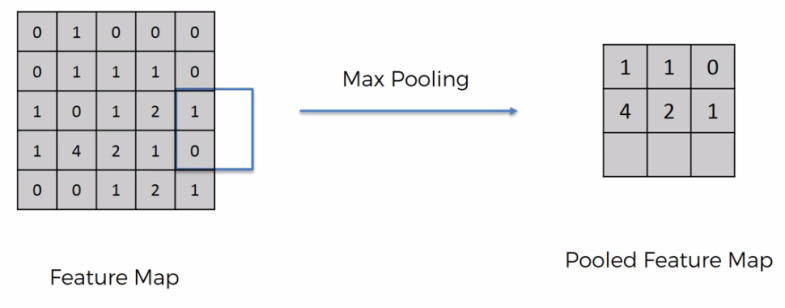
# *step 1 : Convolution - layer*

cnn\_classifier**.add**(**Convolution2D**(32, 3, 3, input\_shape = (64, 64, 3), activation= "relu"))

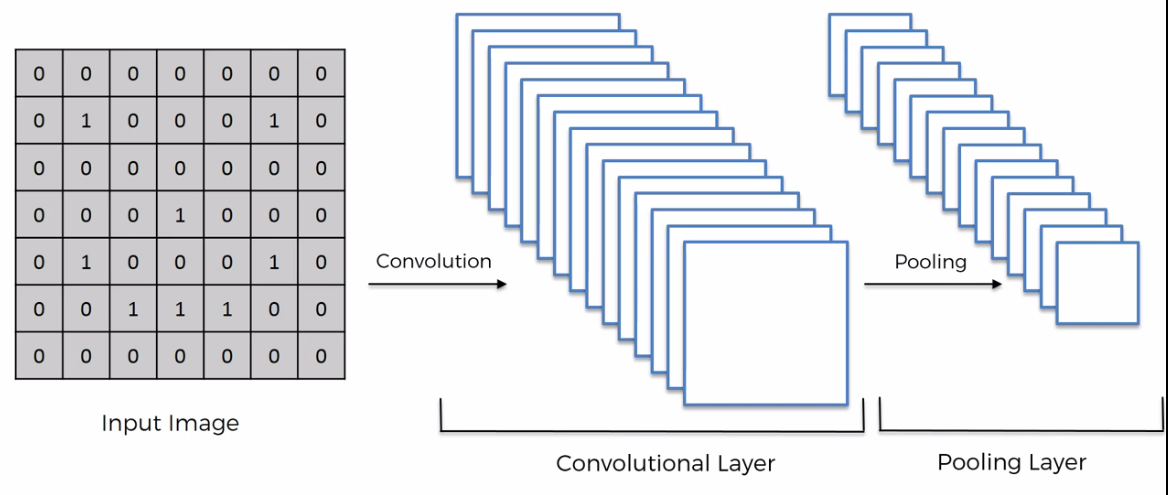
**9.5.7** **Adding Layers: Step 2 : Pooling - layer**

We are ready to move on to 2nd Step, Pooling step. This pooling step is very easy, it just consists of *reducing* the *size* of your *feature maps*.

* In Previous convolution step we've used a ***px*** size *table/filter* for *convolution operation*. We've also use ***Stride-size*** of ***1*** (the filter moves ***1px*** at a time, with 2px overlapping).
* Now in this pooling step we're gonna use a ***px*** size *table* for *Max-pooling operation* (choosing ***maximum value*** of the ***four*** ***cells*** inside this table/square). Now we're gonna use ***Stride-size*** of ***2*** (the table moves ***2px*** at a time, with no overlapping).



* Since *each* *time* we take the *max* of a *2-by-2 table*, at the *end*, we get a *new feature map (pooled-feature-map)*, with a *reduced* *size*. And more precisely, the *size* of the *pooled* *feature* *map* will be the *half* of the *original feature-map's size*.
* And then we obtain our next layer *composed* of all these *pooled-feature-maps* and that is called the Pooling Layer.
* This will reduce the complexity and the *time execution* but without the *losing* too much *information* because we are using max-pooling.
* Using max-pooling we are *keeping* *track* of the *most-important features* of the image, that contained the *high* *numbers* corresponding to where the *feature* *detectors*, detected some *specific* *features* in the input image.



|  |  |
| --- | --- |
| * Remember, we apply this *max* *pooling* to reduce the *number of nodes* for the *Flattening* step, it is some kind of *reduction of independent variables*. * Here we are actually *reducing the size of*  flattened one-dimensional vector (which will be huge) for the Full Connection step. * If we don't reduce the size of these feature maps, we'll get to too large flattened one-dimensional vector and then we'll get too many nodes in the *fully connected layers* in the *NN* part and therefore our model will be highly *compute-intensive*. * So we don't lose the ***spatial structure*** information, hence we don't lose the performance of the model. * But at the same time, we managed to reduce the *time* *complexity* and we make it *less compute-intensive*. | |
|  |  |

* Creating Pooling-layer: We'll apply same ***add()*** method to our classifier ***cnn\_classifier*** to add a pooling-layer. And inside ***add()*** we use ***MaxPooling2D*** :
* Parameters of ***MaxPooling2D***:
* **pool\_size:** We are gonna use ***pool\_size = (2, 2)***. Most of the time we take a ***2-by-2*** pool\_size when we apply max pooling on our feature maps. Because we don't wanna lose the information. We're still being precise on where we have the high numbers (max-pooling) in the feature maps.
* Basically adding following line will *reduce* the *size* of your *feature* *maps*, and *pooled-feature-map's* *size* will be half of *original-feature-map's* size.

# *step 2 : Pooling  - layer*

**cnn\_classifier.add**(**MaxPooling2D**(pool\_size = (2, 2)))

So we just reduced the *complexity* of our model without reducing its performance.

**9.5.8** **Adding Layers: Step 3 : Flattening for ANN-part**

|  |  |
| --- | --- |
| Flattening step consists of taking all our Pooled Feature Maps and put them into one single vector.   * This is going to be a *huge vector* of course because even if we *reduced* the size of the *Feature Maps* in *pooled-feature-maps*, we still have ***many Pooled Feature Maps***. * This *single vector* is going to be the *Input Layer* of a future *ANN*, this *ANN* is similar to the previous chapter's *ANN* (which had *hidden* *layers*), but this *ANN* has *Fully Connected Layers* (hidden-layers with full connection with all nodes).   Now we can ask ourselves two very important questions: |  |

* *Why don't we lose the* spatialstructure *by* Flattening *all these* Feature Maps *into one same,* single vector*?*
* By creating *convolution-layer* of *feature-maps*, we extracted the *spatial structure information*, by getting some *high numbers* in *each Feature Map*, using specific *Feature-Detector/filter* that we applied on the Input Image.
* These *high* *numbers* in each *Feature* *Map* represent the *spatial structure* of our images, because these *high* *numbers* are associated to a *specific* *feature* in the input image.
* When we apply the *Max* *Pooling* Step, we *keep* these *high numbers* because we take the *Max-values*.
* The *Flattening Step* just consists of putting all the *numbers* in the cells of each *pooled-Feature-Map* of the *pooling-layer* into one, same, single *vector*. Hence those *high numbers* that are associated to a *specific* *feature* in the *input* *image* (represents the *spatial* structure) are *still* in the flattened-1D-vector.

Hence all the spatial structure information preserved in this one, huge, single flattened-1D-vector.

* *Why didn't we* directly *take all the* pixels *of the* Input Image *and* Flatten *them into this one, same,* single vector*, without applying the previous steps:* Convolution *and* Max-Pooling*?*
* If we directly *Flatten* all the *Input* *Image* *pixels* into this huge, single, *one-dimensional vector*, then each node of this huge vector will represent *one independent pixel* of the Image.
* Then we only get information of the *pixel* *itself* not the other *pixels* that are *around* it.
* We don't get information of how this pixel is spatially connected to the other pixels around it. We don't get any information of the *spatial structure* around this pixel.
* If we apply the Convolution and the Max Pooling step to create all the *pooled-Feature-Maps*, and *Flatten* all these Maps into this huge, single, one-dimensional vector:
* Then since each *Feature* *Map* corresponds to one *specific* *feature* of the image, then *each* *node* of this *huge* flattened-1D-vector will represent the *information* of a *specific* *feature*, a *specific* *detail* of the *Input* *Image*, (for example, the upper left border of a dog nose).
* Because this *high* *number* doesn't represent a *unique* *pixel* by *itself* but the study of *specific* *feature*, that the Feature Detector extracted from the Input Image, through the Convolution Operation.

And therefore eventually, we keep the spatial structure information of the Input Image.

* Flattening the Pooling-layer: As usual, we're gonna take our classifier ***cnn\_classifier***, apply same ***add()*** method to our classifier and inside ***add()*** we use ***Flatten()*** to flatten the pooling-layer.

# *step 3 : Flattening*

**cnn\_classifier.add**(**Flatten**())

* There is no need to specify any parameters, because ***Keras*** will understand that,
* Since we're taking a ***cnn\_classifier*** object, that it needs to Flatten the previous layer that we obtain in the Max Pooling Step, after the Convolution Step.

Now this huge, single vector is created and basically it contains all the information of the spatial structure of our images. Then the 4th step is to create a *classic* *ANN*, that will classify the images using this *huge, single vector*, as the *Input* *Vector*.

**9.5.9** **Adding Layers: Step 4 : ANN – Full Connection**

The *full* *connection* step, which basically consists of making a classic *ANN* composed of some *fully-connected layers* (instead of hidden layer).

* We managed to *convert* our *input* *image* into this *one-dimensional vector* that contains some information of the *spatial* *structure* or of some *pixel patterns* in the image.
* We use this input vector as the input layer of a classic ANN. A classic ANN, can be a great classifier for ***nonlinear problem*** like ***image*** ***classification***.
* Since we already have our ***input layer (flattened-vector)***, now we create a hidden layer i.e a fully-connected layer.
* Fully-connected-layer: This is similar to previous chapter, but output-layer will have two nodes.
* We use ***Dense()*** method inside ***cnn\_classifier.add()*** to add the fully-connected layer (hidden layer).
* Parameters:

1. ***units:*** is the number of nodes in the hidden layer. *How many nodes do we need to input here?*

* Because, there was *no rule of thumb* to choose a *number* *of* *nodes* in the *hidden* *layer*.
* We saw that a common practice is to choose a number of hidden nodes *between* the *number of input nodes* and the *number of output* *nodes*.
* But here we have too many input nodes, because, we built ***32*** pooled-feature-maps and each contains many *cells*, those are contained in *flattened-one-single-vector* which is the *input* *layer* of our *fully-connected layer*. Thus we end up with a lot of input nodes.
* So we are not gonna count all of them right now, just remember that we *shouldn't take* a *too-small number*.
* We are gonna ***choose*** here ***128***.
* Remember this choice of numbers *results* from *experimentation*. **128** is not so small and not too big to make it highly *compute-intensive*.
* By experimenting on this outputting parameter, we realize that a number around 100 is a good choice. We could have picked 100, but it is a common practice to pick a power of 2 (i.e ).

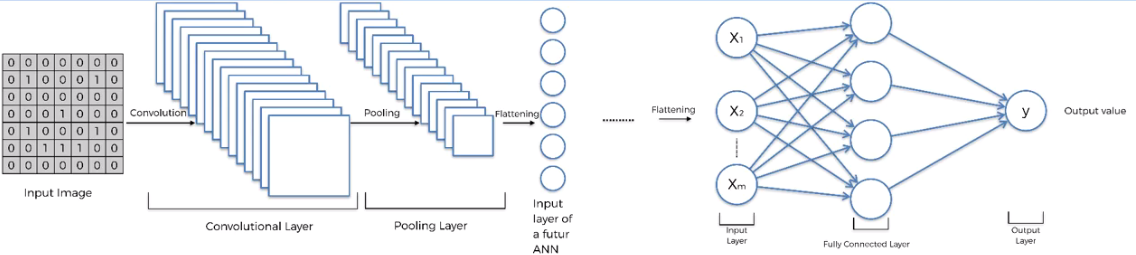
1. ***activation***: The activation function, will be "relu" for the fully connected layers ***activation= "relu"***.

**cnn\_classifier.add**(**Dense**(units= 128, activation= "relu")) # *fully connected layers*

* Output-layer: The last layer that we need to add is the output layer. The value of ***output\_dim = 1*** because out output will be cat/dog. We are just expecting one node that is going to be the predicted probability of one class, the dog or the cat.
* The activation function is the SIGMOID activation function. Because we have a binary outcome, cats or dog.
* If we had an outcome with *more* than *two* *categories*, we would need to use the SOFTMAX activation function.

**cnn\_classifier.add**(**Dense**(units= 1, activation= "sigmoid")) # *output layer*

* This full connection step only consisted of adding the *fully-connected layer*, that is the *hidden layer*, and then the *output layer* to get the *final* *predictions*.



# *step 4 : ANN - full connection*

**cnn\_classifier.add**(**Dense**(units= 128, activation= "relu")) # *fully connected layers*

**cnn\_classifier.add**(**Dense**(units= 1, activation= "sigmoid")) # *output layer*

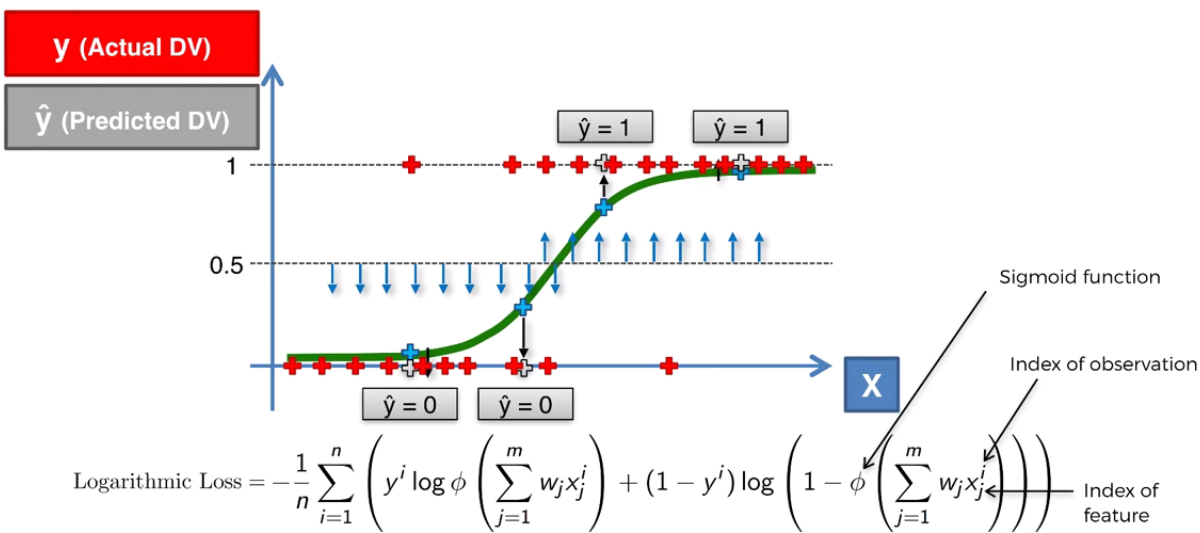
**9.5.10** **Compilation: compile - CNN model**

We just need to compile the whole NN by choosing a *SGD* *algorithm*, a *loss* *function* and, eventually, a *performance* *metric*. It is same as the *classic-ANN* in the *previous* *chapter*.

# *compile the NN*

**cnn\_classifier.compile**(optimizer= "adam", loss="binary\_crossentropy", metrics = ["accuracy"])

* We apply ***compile()*** over our classifier ***cnn\_classifier***.
* Parameters:
* optimizer: " ***adam*** " specifies the ***SGD algorithm***. It is the Adam algorithm.
* loss: We choose the ***binary\_crossentropy*** for two reasons:
* First of all, because this function corresponds to the logarithmic loss, that is the loss function that we use in general for *classification* problems using a *classification* *model* like *logistic* *regression*.
* The second reason is that we have a binary outcome, cat or dog. and therefore, we need to choose the binary-cross-entropy loss function.
* If we had more than two outcomes, like cats, dogs, and birds, well, we would need to choose categorical-cross-entropy as loss function.
* metrics: To choose the *performance metric*. The most common performance metric is the accuracy metric.



**9.5.11** **Image Preprocessing**

We just completed ***part one: Building the CNN***. We designed the architecture of our CNN.

* Now we're beginning ***part two: Image Preprocessing***, where we will *fit our CNN to our images*.
* We will actually do it in one step because we're gonna use a shortcut, the ***Keras Documentation***.
* We will use ***Keras Documentation*** for a process called image augmentation, that basically consists of ***preprocessing*** your ***images*** to prevent ***overfitting***.
* If we don't do this Image Augmentation, we might get is a *great* *accuracy* result on the *training* *set*, but a much *lower* *accuracy* on the *test* *set*.
* Image Augmentation Process: We know that one of the situations that lead to overfitting is when we have *few data* to *train* our *model*. In that situation, our model finds some *correlations* in the *few observations* of the *training* *set*, but *fails* to generalize this *correlations* on some *new observations*.
* And when it comes to images, we actually need a lot of images to find and generalize some correlations, because in Computer Vision, our model doesn't need to find some *correlations* between *independent and dependent variables*. It needs to find some *patterns* in the *pixels*, and to do this it requires a *lot* of *images*.
* Right now, we are working with 10,000 images, 8,000 images on the training set, and that is actually not much to get some great performance on results. We either need some *more images*, or we can use *data augmentation* trick.

|  |
| --- |
| * Image augmentation will create many batches of our images, and in each batch it will apply some random transformations on a *random* *selection* of our *images*, like ***rotating*** them, ***flipping*** them, ***shifting*** them, or even ***shearing*** them, and eventually we'll get many more *diverse* *images* inside these *batches*, and therefore a lot *more* *material* to *train*. * That's why it is called *image augmentation*. That's because the *amount* of our training images is *augmented*. Besides, because of the *random transformations*, our model will *never* find the *same picture* across the *batches*. * In summary, image augmentation is a technique that allows us to enrich our data set, *without* *adding* more *images* and therefore that allows us to *get good performance* results with *little or not overfitting*, even with a *small* *amount* of *images*. |

* We're gonna use this TensorFlow-Keras Documentation shortcut. In your browser, you can type *"keras image preprocessing tensorflow"*. Search the link for Documentation.
* We will go to the link "imagedatagenerator": However the code might change in latest version. Now we use following link:

<https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator>

The Keras Library Documentation is also available currently on:

<https://faroit.com/keras-docs/1.2.0/preprocessing/image/>

* Open **tf.keras**. You get a lot of *informations* about *Keras* and ready-to-use codes that you can take for your deep learning project.

|  |  |
| --- | --- |
| **tf.keras.preprocessing.image.ImageDataGenerator**(      featurewise\_center=**False**,      samplewise\_center=**False**,      featurewise\_std\_normalization=**False**,      samplewise\_std\_normalization=**False**,      zca\_whitening=**False**,      zca\_epsilon=1e-06,      rotation\_range=0,      width\_shift\_range=0.0,      height\_shift\_range=0.0,      brightness\_range=**None**,      shear\_range=0.0,      zoom\_range=0.0,      channel\_shift\_range=0.0,      fill\_mode='nearest',      cval=0.0,      horizontal\_flip=**False**,      vertical\_flip=**False**,      rescale=**None**,      preprocessing\_function=**None**,      data\_format=**None**,      validation\_split=0.0,      interpolation\_order=1,      dtype=**None**  ) |  |

|  |  |
| --- | --- |
| * We're gonna look for Preprocessing. * Note that, deep learning can also be applied to text in a very powerful way. There we use " text preprocessing ". * ImageDataGenerator: That's the first function that we're gonna use to *generate* this *image augmentation*. * From following page we'll take ready-to-use code and that corresponds very well to how we structured our data set,   <https://faroit.com/keras-docs/1.2.0/preprocessing/image/> |  |

* There are *two ways* to *preprocess* our *images* by applying *image augmentation* on them:
* It's either by using this code that is based on the flow method:

Example of using .**flow**(X, y): Following doesn't match to our *folder-structure*.

(X\_train, y\_train), (X\_test, y\_test) = **cifar10.load\_data**()

Y\_train = **np\_utils.to\_categorical**(y\_train, nb\_classes)

Y\_test = **np\_utils.to\_categorical**(y\_test, nb\_classes)

datagen = **ImageDataGenerator**(

    featurewise\_center=**True**,

    featurewise\_std\_normalization=**True**,

    rotation\_range=20,

    width\_shift\_range=0.2,

    height\_shift\_range=0.2,

    horizontal\_flip=**True**)

# *compute quantities required for featurewise normalization*

# *(std, mean, and principal components if ZCA whitening is applied)*

**datagen.fit**(X\_train)

# *fits the model on batches with real-time data augmentation:*

**model.fit\_generator**(**datagen.flow**(X\_train, Y\_train, batch\_size=32),

                    samples\_per\_epoch=len(X\_train), nb\_epoch=nb\_epoch)

# *here's a more "manual" example*

**for** e **in** **range**(nb\_epoch):

**print** 'Epoch', e

    batches = 0

**for** X\_batch, Y\_batch **in** **datagen.flow**(X\_train, Y\_train, batch\_size=32):

        loss = **model.train**(X\_batch, Y\_batch)

        batches += 1

**if** batches **>=** **len**(X\_train) / 32:

            # *we need to break the loop by hand because*

            # *the generator loops indefinitely*

**break**

* Or this code that is based on the **flow\_from\_directory** method. Example of using .**flow\_from\_directory**(directory):
* Following matches to our *folder-structure*. We'll use this code section because we structured our folder in this specific way so that our image classes of cat or dogs can be well identified in the separate folders.

train\_datagen = **ImageDataGenerator**(

        rescale=1./255,

        shear\_range=0.2,

        zoom\_range=0.2,

        horizontal\_flip=**True**)

test\_datagen = **ImageDataGenerator**(rescale=1./255)

train\_generator = **train\_datagen.flow\_from\_directory**(

        'data/train',

        target\_size=(150, 150),

        batch\_size=32,

        class\_mode='binary')

validation\_generator = **test\_datagen.flow\_from\_directory**(

        'data/validation',

        target\_size=(150, 150),

        batch\_size=32,

        class\_mode='binary')

**model.fit\_generator**(

        train\_generator,

        samples\_per\_epoch=2000,

        nb\_epoch=50,

        validation\_data=validation\_generator,

        nb\_val\_samples=800)

* Since our dataset is on our working directory, we need to change few things on ***flow\_from\_directory()*** function.
* Basically above *copied code-segment* has everything that we need to preprocess-augment our images, and even fitting our CNN that we just built on our images.
* That's the end of the code, because this ***fit\_generator()*** method will not only fit our ***CNN*** to the ***training\_set***, but at same time it will also *test* its *performance* on some *new observations* of our test set, (i.e the images of our ***test\_set*** folder).
* First we need to import ***ImageDataGenerator*** fronn ***keras***.
* We'll create ***two*** ***data-generator/augmentation objects*** for train and test data named ***train\_datagen*** and ***test\_datagen***.
* For those 2 objects we specify the *transform* *parameters*. In this *image* *augmentation* part, we apply several transformations like :

1. **rescale:** rescaling factor. Defaults to None. If **None** or **0**, *no rescaling* is applied, otherwise we multiply the data by the value provided (before applying any other transformation).

* ***rescale*** is always compulsory and it corresponds to the feature scanning part of the data preprocessing phase that we know.
* Why we have to rescale by 1. / 255: ***rescale*** is a value by which we will ***multiply the data*** before any ***other*** ***processing***. Our original images consist in ***RGB*** ***coefficients*** in the ***0-255***, but such values would be too high for our models to process (given a typical learning rate), so we target values between ***0*** and ***1*** instead by scaling with a **1./255** factor.

1. **shear\_range:** (Float). Shear Intensity (Shear angle in counter-clockwise direction as radians).

* **sharing** is a ***geometrical*** ***transformation*** that is also called ***transvection***. Here the pixels are moved to a fixed direction over a proportional distance from a line that is parallel to the direction they're moving to. So basically that is just a *geometrical transformation* for augmenting our images.

1. **zoom\_range:** (Float) Range for random zoom. If a float, [lower, upper] = [1-zoom\_range, 1+zoom\_range]. This is some sort of *random zoom* that we apply on our images.
2. **horizontal\_flip:** (Boolean) Randomly flip inputs-images horizontally.

However we can also have ***vertical***\_***flip***, but that is not used here.

We can have fun and apply all the image transformations that there are in this Keras Documentation, but for now we will just use what we have in this code segment. That will be way enough and you'll see that we get good results.

* Rename ***train\_generator*** and ***validation\_generator*** to ***training\_set*** and ***test\_set*** respectively. These two are the instances of objects ***train\_datagen*** and ***test\_datagen*** and we apply ***flow\_from\_directory*** method to import the train and test images to apply augmentation and then CNN on them. We need to *specify* following *parameters* for ***flow\_from\_directory***

1. **directory:** path to the *target* *directory*. It should contain one *subdirectory* *per* *class*. Any PNG, JPG or BNP images inside each of the subdirectories directory tree will be included in the generator.

* We have to replace "directory" with corresponding file path. Set the image folder paths **'dataset/training\_set'** and **'dataset/test\_set'** for *training\_set* and *test\_set* respectively. (notice back slash **/** used instead of **\**)
* We *don't* have to *specify* the *whole* *path* that leads to this ***dataset***, because this dataset is already in the *working* *directory* folder.

1. **target\_size:** *Tuple* of integers, default: ***(256, 256)***. The *dimensions* to which all images found will be *resized*. In our case since we are working with *CPU*, we set it to ***(64, 64)*** it should be same as we set ***input\_shape*** (dimension of expected resized images) in ***Convolution2D***.
2. **batch\_size:** Size of the batches of data (default: 32). ***batch\_size*** is not related to no. of filters used in ***Convolution2D***.

* It specifies the ***number*** of ***random samples*** of our images that will go through the CNN, after which the weight will be updated.

1. **class\_mode:** one of "categorical", "binary", "sparse" or None. Default: "categorical". Here we use ***"binary"*** because we are working with *binary* *category* cat/dog.

* That's the parameter *indicating* if your *class*, your *dependent* *variable*, is ***binary*** or has ***more than two categories***, and therefore since we have two classes here, cats and dogs, well the **class\_mode = "binary"**.
* ***class\_mode*** Determines the ***type*** of ***label arrays*** that are returned:
* "***categorical***" will be ***2D*** ***one-hot encoded labels***,
* "***binary***" will be ***1D binary labels***,
* "***sparse***" will be ***1D integer labels***.
* If ***None***, ***no labels*** are returned.
* These two sections actually create the training-set and the test-set. Basically in this section we will create this ***training\_set*** composed of all these ***augmented*** ***images*** extracted from our ***ImageDataGenerator***.
* Also ***test\_set*** will create our test set from the images of the *test\_set folder* that are extracted from our *ImageDataGenerator*.
* ***test\_set*** will be used to ***evaluate*** the ***model*** ***performance*** in ***fit\_generator()*** part of the code.
* Finally last code section, the ***model.fit\_generator***, where we ***fit*** our ***CNN*** to the ***training set***, while also ***testing*** its ***performance*** on the ***test*** ***set***. Replace "model" with " **cnn\_classifier**".
* ***fit\_generator()*** at the end of the code section, will fit our ***CNN*** model on the ***training\_set***, as well as *testing* its *performance* on the ***test\_set***.
* We are using this **fit\_generator()** method to fit our CNN to our ***training\_set*** and test its performance on the ***test\_set*** at the same time, and this **fit\_generator()** method is applied onto our CNN model. We named our model as **cnn\_classifier.**

1. The first argument is our ***training\_set***.
2. ***samples\_per\_epoch:*** Is the number of images we have in our training set. Remember all the observations of the training set pass through the CNN during each epoch, and since we have 8,000 images in our training set, we set ***samples\_per\_epoch = 8000***.
3. ***nb\_epoch:*** That's the number of epochs we wanna choose to train our CNN. And here, 25 might be a good choice. So that we don't have to wait for too long to get our results.
4. ***validation\_data:*** It corresponds to the ***test\_set***, on which we want to evaluate the performance of our CNN, so we set ***validation\_data = test\_set.***
5. ***nb\_val\_samples:*** It corresponds to the number of images in our test\_set, and that is 2,000.

# *---------- Part 2 - Image Preprocessing & fit CNN to our images ----------------*

**from** keras.preprocessing.image **import** ImageDataGenerator

# *creating two data-generator/augmentation obgects for train and test data*

        # *here we specify the transform parameters*

train\_datagen = **ImageDataGenerator**(

                                    rescale=1./255,

                                    shear\_range=0.2,

                                    zoom\_range=0.2,

                                    horizontal\_flip=**True** )

test\_datagen = **ImageDataGenerator**(rescale=1./255)

# *applying augmentation on training data : training folder path needed*

                                       # *target\_size = input\_shape (dimension of expected resized images)*

                                       # *batch\_size is not related to no. of filters*

training\_set = **train\_datagen.flow\_from\_directory**('dataset/training\_set',

                                                target\_size=(64, 64),

                                                batch\_size=32,

                                                class\_mode='binary')

# *applying augmentation on test data : test folder path needed*

test\_set = **test\_datagen.flow\_from\_directory**('dataset/test\_set',

                                            target\_size=(64, 64),

                                            batch\_size=32,

                                            class\_mode='binary')

**cnn\_classifier.fit\_generator**(training\_set,

 # *due to incompatibility between Tensorflow and Keras version, "samples\_per\_epoch", "nb\_epoch", "nb\_val\_samples" may not work*

 # *Then use "steps\_per\_epoch", "epochs",  "validation\_steps" instead*

                            # *samples\_per\_epoch = 8000,*

                            # *nb\_epoch=25,*

                            # *nb\_val\_samples = 2000,*

                            steps\_per\_epoch = 250,

                            epochs = 25,

                            validation\_data=test\_set,

                            validation\_steps = 62)

* Note: Due to ***incompatibility*** between ***Tensorflow*** and ***Keras*** version, "**samples\_per\_epoch**", "**nb\_epoch**", "**nb\_val\_samples**" may not work.
* Then use "**steps\_per\_epoch**", "**epochs**", "**validation\_steps**" instead.
* Remember in image-augmentation we used ***32*** as ***batch\_size*** in ***training\_set***. Then we need to set **steps\_per\_epoch= 250**. Because we are dividing our dataset into several batches. And **32\*250 = 8000**. Similarly **validation\_steps= 62** because ***test\_data*** size is ***2000*** we have 200 test image and we also used we used ***32*** as ***batch\_size*** in ***test\_set*** (**32\*62 = 1984**).
* Errors: Make sure that your dataset or generator can generate at least "**steps\_per\_epoch \* epochs**" batches

Epoch 1/25

25/32 [======================>.......] - ETA: 1s - loss: 0.6942 - accuracy: 0.5238WARNING:tensorflow:Your input ran out

of data; interrupting training. Make sure that your dataset or generator can generate at least "**steps\_per\_epoch \* epochs**" batches (in this case, 800 batches). You may need to use the ***repeat()*** function when building your dataset.

* To avoid this, instead of manually setting "**steps\_per\_epoch**", "**epochs**", "**validation\_steps**": we use following code:

**cnn\_classifier.fit\_generator**(training\_set,

                            steps\_per\_epoch=**math.floor**((training\_set.samples)/(training\_set.batch\_size)),

                            epochs=25,

                            validation\_data=test\_set,

                            validation\_steps=**math.floor**((test\_set.samples)/(test\_set.batch\_size)),

                                  )

|  |  |
| --- | --- |
| * After the training is over. We obtained an ***accuracy*** of ***84%*** for the ***training set***, and ***75%*** for the ***test set***. Well, not too bad, but not too good either. * The *difference* between the *accuracy* of the *training set* and the *test set* indicating whether there's *overfitting* or not. * So *75%* accuracy on the *test set* is not bad. That means that we get *three* *correct* *predictions* out of *four*, so that's actually not too bad. * When we get quite a large difference between the ***accuracy on the training set*** and the ***accuracy on test set***. It's indicating that there is important overfitting. | * Notice how ***test\_set*** and ***training\_set*** classifies the images: |

* Now we'll improve our model ***by*** adding an ***extra layer***, to make this ***accuracy*** on ***test-set*** will reach an accuracy ***over 80%*** and decrease this difference between the ***training\_set accuracy*** and the ***test\_set accuracy***.

**9.5.12** **Increasing Accuracy**

To improve our model's accuracy we need make a deeper NN that is a deeper CNN.

* We have two option:

1. First option is to add another ***convolutional*** ***layer*** and ***pooling*** ***layer***.
2. Second option is to *add* another *fully connected layer*(hidden-layer).

* The best solution is actually to add a *convolutional* *layer*. And its very easy. We have to add just 2-lines of code, after our *first pooling* *layer*. But you can always improve your model by considering the two options that is adding a convolutional layer as well as a fully connected layer.
* However, we can to reach our goal of getting a test set accuracy of more than 80% (also *reducing* the *over-fitting*) by only adding a second convolutional layer.
* We apply the *2nd convolution operation* to this *first pooling layer*.
* So we build this *2nd convolution* and *2nd pooling layers* right after the first two layers, i.e. after *1st convolution* and *1st pooling layers* .
* And before the flattening step.

# *step 1 : Convolution - layer*

**cnn\_classifier.add**(**Convolution2D**(32, 3, 3, input\_shape = (64, 64, 3), activation= "relu"))

# *step 2 : Pooling  - layer*

**cnn\_classifier.add**(**MaxPooling2D**(pool\_size = (2, 2)))

# *improving step : Adding 2nd-Convolution and  2nd-Pooling layers*

**cnn\_classifier.add**(**Convolution2D**(32, 3, 3, activation= "relu"))

**cnn\_classifier.add**(**MaxPooling2D**(pool\_size = (2, 2)))

|  |
| --- |
| * Notice, we don’t need ***input\_shape*** parameter, because there is no new *input-images* for this *layer*, we are just taking the pooled-feature-maps coming from the previous step the first-pooling-layer. * So we're going to apply the *convolution* trick and the *max pooling* trick, *not on the images* but *on the pooled feature maps*. * Hence we don’t need ***input\_shape***. Because ***input\_shape*** corresponds to our images dimensions that our CNN should expect. |

* Therefore when you're adding an additional convolutional layer, you just need
* a number *of features detectors*,
* the dimensions of these *feature detectors*
* and an activation function.
* And then you apply max pooling with only pool\_size parameter.
* If you want to have fun adding new additional convolutional layers, then you can ***increase*** the number of ***feature*** ***detectors*** and ***double*** it ***each*** ***time***.
* So for example, you can add a ***third*** ***convolutional*** ***layer*** with ***64 feature detectors***. That's a common practice and that leads to great results.
* After ***25*** ***epoch***, are about to get an accuracy of ***85%*** for the ***training*** ***set*** and ***82%*** for the ***test*** ***set***.
* That's great!! We're not only reached our goal to obtain the test set accuracy over 80% and also we *reduced* the *difference* between the *training set accuracy* and the *test set accuracy*. Because now indeed we get a ***difference*** of ***3%*** as opposed to this ***10%*** difference that we got in the ***previous*** ***result***. (***84%*** on ***training*** ***set*** and ***75%*** on ***test*** ***set***)

|  |
| --- |
| * Getting even-more accuracy: Of course, adding more convolutional layers will help get an even better accuracy. But if you increase more, ***increase*** the ***input\_shape*** of image. * Higher image-size: Using higher ***input\_shape*** & ***target\_size*** gives a better accuracy for your images of the *train set* and the *test set* so that you get more information of your pixel patterns. * Because if you increase the size of your images, all your images will be resized, you will get *a lot more pixels* in the *rows* and a *lot more pixels* in the *columns* in your *input* *images*, therefore you will have more information on the pixels. * GPU and more time: To do this, we recommend using a GPU or trying this before getting to sleep and you might even be able to get an accuracy **over 90%.** |

***Training-set accuracy & test-set accuracy:***

Epoch 1/100

250/250 [==============================] - 86s 340ms/step - loss: 0.6881 - **accuracy: 0.5411** - val\_loss: 0.6721 - **val\_accuracy: 0.5796**

* ***accuracy: 0.5411*** is the accuracy on ***training\_set***.
* ***val\_accuracy: 0.5796*** is the accuracy on ***test\_set***

**Practiced version**

# *=========== ====  Convolutional Neural Network : CNN  ==== ============*

# *----------- Install following  packages -------------*

    # *Install Theano*

    # *Install Tensorflow*

    # *Install Tensorflow*

    # *Install Keras*

# *---------- Part 1 - Building the CNN ----------------*

# *Importing the Keras libraries and packages*

**from** keras.models **import** Sequential     # *to initialize as sequence-of-layers*

**from** keras.layers **import** Convolution2D  # *Convolution step for images*

**from** keras.layers **import** MaxPooling2D   # *not "MaxPool2D". Pooling step for images*

**from** keras.layers **import** Flatten        # *Flatenning step*

**from** keras.layers **import** Dense          # *ads fully-connected-layers to classic ANN*

# *initializing the CNN*

cnn\_classifier = **Sequential**()

# *step 1 : Convolution - layer*

**cnn\_classifier.add**(**Convolution2D**(32, 3, 3, input\_shape = (64, 64, 3), activation= "relu"))

# *step 2 : Pooling  - layer*

**cnn\_classifier.add**(**MaxPooling2D**(pool\_size = (2, 2)))

# *improving step : Adding 2nd-Convolution and  2nd-Pooling layers*

**cnn\_classifier.add**(**Convolution2D**(32, 3, 3, activation= "relu"))

**cnn\_classifier.add**(**MaxPooling2D**(pool\_size = (2, 2)))

# *step 3 : Flattening*

**cnn\_classifier.add**(**Flatten**())

# *step 4 : ANN - full connection*

**cnn\_classifier.add**(**Dense**(units= 128, activation= "relu")) # *fully connected layers*

**cnn\_classifier.add**(**Dense**(units= 1, activation= "sigmoid")) # *output layer*

# *compile the NN*

**cnn\_classifier.compile**(optimizer= "adam", loss="binary\_crossentropy", metrics = ["accuracy"])

# *---------- Part 2 - Image Preprocessing & fit CNN to our images ----------------*

**from** keras.preprocessing.image **import** ImageDataGenerator

**import** math

# *creating two data-generator/augmentation obgects for train and test data*

        # *here we specify the transform parameters*

train\_datagen = **ImageDataGenerator**(

                                    rescale=1./255,

                                    shear\_range=0.2,

                                    zoom\_range=0.2,

                                    horizontal\_flip=**True** )

test\_datagen = **ImageDataGenerator**(rescale=1./255)

# *applying augmentation on training data : training folder path needed*

                                        # *target\_size = input\_shape (dimension of expected resized images)*

                                        # *batch\_size is not related to no. of filters*

training\_set = **train\_datagen.flow\_from\_directory**('dataset/training\_set',

                                                target\_size=(64, 64),

                                                batch\_size=32,

                                                class\_mode='binary')

# *applying augmentation on test data : test folder path needed*

test\_set = **test\_datagen.flow\_from\_directory**('dataset/test\_set',

                                            target\_size=(64, 64),

                                            batch\_size=32,

                                            class\_mode='binary')

"""

cnn\_classifier.fit\_generator(training\_set,

 # due to incompatibility between Tensorflow and Keras version, "samples\_per\_epoch", "nb\_epoch", "nb\_val\_samples" may not work

 # Then use "steps\_per\_epoch", "epochs",  "validation\_steps" instead

                            # samples\_per\_epoch = 8000,

                            # nb\_epoch=25,

                            # nb\_val\_samples = 2000,

                            steps\_per\_epoch = 250,  # training\_set\_size/batch\_size

                            epochs = 25,

                            validation\_data=test\_set,

                            validation\_steps = 62   # test\_set\_size/batch\_size

                            )

"""

*# In this case no need to explicitly specify training\_set's or test\_set's sample-size: i.e 8000, 2000 or 800, 200*

**cnn\_classifier.fit\_generator**(

                            training\_set,

                            steps\_per\_epoch=**math.floor**((training\_set.samples)/(training\_set.batch\_size)),

                            epochs=25,

                            validation\_data=test\_set,

                            validation\_steps=**math.floor**((test\_set.samples)/(test\_set.batch\_size))

                            )

# *history = model.fit\_generator(train\_gen,*

# *steps\_per\_epoch=(train\_gen.samples/batch\_size),  # len(train\_gen)*

# *epochs=100,*

# *validation\_data=validation\_gen,*

# *validation\_steps=(validation\_gen.samples/batch\_size),*

# *callbacks=[checkpointer],*

# *workers=4*

# *)*

# *python prctc\_cnn.py*

|  |
| --- |
| ImageDataGenerator **keras.preprocessing.image.ImageDataGenerator**(featurewise\_center=**False**,      samplewise\_center=**False**,      featurewise\_std\_normalization=**False**,      samplewise\_std\_normalization=**False**,      zca\_whitening=**False**,      rotation\_range=0.,      width\_shift\_range=0.,      height\_shift\_range=0.,      shear\_range=0.,      zoom\_range=0.,      channel\_shift\_range=0.,      fill\_mode='nearest',      cval=0.,      horizontal\_flip=**False**,      vertical\_flip=**False**,      rescale=**None**,      dim\_ordering=**K.image\_dim\_ordering**())  Generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches) indefinitely.   * **Arguments**:   + **featurewise\_center**: Boolean. Set input mean to 0 over the dataset, feature-wise.   + **samplewise\_center**: Boolean. Set each sample mean to 0.   + **featurewise\_std\_normalization**: Boolean. Divide inputs by std of the dataset, feature-wise.   + **samplewise\_std\_normalization**: Boolean. Divide each input by its std.   + **zca\_whitening**: Boolean. Apply ZCA whitening.   + **rotation\_range**: Int. Degree range for random rotations.   + **width\_shift\_range**: Float (fraction of total width). Range for random horizontal shifts.   + **height\_shift\_range**: Float (fraction of total height). Range for random vertical shifts.   + **shear\_range**: Float. Shear Intensity (Shear angle in counter-clockwise direction as radians)   + **zoom\_range**: Float or [lower, upper]. Range for random zoom. If a float, [lower, upper] = [1-zoom\_range, 1+zoom\_range].   + **channel\_shift\_range**: Float. Range for random channel shifts.   + **fill\_mode**: One of {"constant", "nearest", "reflect" or "wrap"}. Points outside the boundaries of the input are filled according to the given mode.   + **cval**: Float or Int. Value used for points outside the boundaries when fill\_mode = "constant".   + **horizontal\_flip**: Boolean. Randomly flip inputs horizontally.   + **vertical\_flip**: Boolean. Randomly flip inputs vertically.   + **rescale**: rescaling factor. Defaults to None. If None or 0, no rescaling is applied, otherwise we multiply the data by the value provided (before applying any other transformation).   + **dim\_ordering**: One of {"th", "tf"}. "tf" mode means that the images should have shape (samples, height, width, channels), "th" mode means that the images should have shape (samples, channels, height, width). It defaults to the image\_dim\_ordering value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "tf". * **Methods**:   + **fit(X)**: Compute the internal data stats related to the data-dependent transformations, based on an array of sample data. Only required if featurewise\_center or featurewise\_std\_normalization or zca\_whitening.     - **Arguments**:       * **X**: sample data. Should have rank 4. In case of grayscale data, the channels axis should have value 1, and in case of RGB data, it should have value 3.       * **augment**: Boolean (default: False). Whether to fit on randomly augmented samples.       * **rounds**: int (default: 1). If augment, how many augmentation passes over the data to use.       * **seed**: int (default: None). Random seed.   + **flow(X, y)**: Takes numpy data & label arrays, and generates batches of augmented/normalized data. Yields batches indefinitely, in an infinite loop.     - **Arguments**:       * **X**: data. Should have rank 4. In case of grayscale data, the channels axis should have value 1, and in case of RGB data, it should have value 3.       * **y**: labels.       * **batch\_size**: int (default: 32).       * **shuffle**: boolean (defaut: True).       * **seed**: int (default: None).       * **save\_to\_dir**: None or str (default: None). This allows you to optimally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).       * **save\_prefix**: str (default: ''). Prefix to use for filenames of saved pictures (only relevant if save\_to\_dir is set).       * **save\_format**: one of "png", "jpeg" (only relevant if save\_to\_dir is set). Default: "jpeg".     - **yields**: Tuples of (x, y) where x is a numpy array of image data and y is a numpy array of corresponding labels. The generator loops indefinitely.   + **flow\_from\_directory(directory)**: Takes the path to a directory, and generates batches of augmented/normalized data. Yields batches indefinitely, in an infinite loop.     - **Arguments**:       * **directory**: path to the target directory. It should contain one subdirectory per class. Any PNG, JPG or BNP images inside each of the subdirectories directory tree will be included in the generator. See [this script](https://gist.github.com/fchollet/0830affa1f7f19fd47b06d4cf89ed44d) for more details.       * **target\_size**: tuple of integers, default: (256, 256). The dimensions to which all images found will be resized.       * **color\_mode**: one of "grayscale", "rbg". Default: "rgb". Whether the images will be converted to have 1 or 3 color channels.       * **classes**: optional list of class subdirectories (e.g. ['dogs', 'cats']). Default: None. If not provided, the list of classes will be automatically inferred (and the order of the classes, which will map to the label indices, will be alphanumeric).       * **class\_mode**: one of "categorical", "binary", "sparse" or None. Default: "categorical". Determines the type of label arrays that are returned: "categorical" will be 2D one-hot encoded labels, "binary" will be 1D binary labels, "sparse" will be 1D integer labels. If None, no labels are returned (the generator will only yield batches of image data, which is useful to use model.predict\_generator(), model.evaluate\_generator(), etc.).       * **batch\_size**: size of the batches of data (default: 32).       * **shuffle**: whether to shuffle the data (default: True)       * **seed**: optional random seed for shuffling and transformations.       * **save\_to\_dir**: None or str (default: None). This allows you to optimally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).       * **save\_prefix**: str. Prefix to use for filenames of saved pictures (only relevant if save\_to\_dir is set).       * **save\_format**: one of "png", "jpeg" (only relevant if save\_to\_dir is set). Default: "jpeg".       * **follow\_links**: whether to follow symlinks inside class subdirectories (default: False). * **Examples**:   Example of using **.flow(X, y)**:  (X\_train, y\_train), (X\_test, y\_test) = **cifar10.load\_data**()  Y\_train = **np\_utils.to\_categorical**(y\_train, nb\_classes)  Y\_test = **np\_utils.to\_categorical**(y\_test, nb\_classes)  datagen = **ImageDataGenerator**(      featurewise\_center=**True**,      featurewise\_std\_normalization=**True**,      rotation\_range=20,      width\_shift\_range=0.2,      height\_shift\_range=0.2,      horizontal\_flip=**True**)  # *compute quantities required for featurewise normalization*  # *(std, mean, and principal components if ZCA whitening is applied)*  **datagen.fit**(X\_train)  # *fits the model on batches with real-time data augmentation:*  **model.fit\_generator**(**datagen.flow**(X\_train, Y\_train, batch\_size=32),                      samples\_per\_epoch=len(X\_train), nb\_epoch=nb\_epoch)  # *here's a more "manual" example*  **for** e **in** **range**(nb\_epoch):  **print** 'Epoch', e      batches = 0  **for** X\_batch, Y\_batch **in** **datagen.flow**(X\_train, Y\_train, batch\_size=32):          loss = **model.train**(X\_batch, Y\_batch)          batches += 1  **if** batches **>=** **len**(X\_train) / 32:              # *we need to break the loop by hand because*              # *the generator loops indefinitely*  **break**  Example of using **.flow\_from\_directory(directory)**:  train\_datagen = **ImageDataGenerator**(          rescale=1./255,          shear\_range=0.2,          zoom\_range=0.2,          horizontal\_flip=**True**)  test\_datagen = **ImageDataGenerator**(rescale=1./255)  train\_generator = **train\_datagen.flow\_from\_directory**(          'data/train',          target\_size=(150, 150),          batch\_size=32,          class\_mode='binary')  validation\_generator = **test\_datagen.flow\_from\_directory**(          'data/validation',          target\_size=(150, 150),          batch\_size=32,          class\_mode='binary')  **model.fit\_generator**(          train\_generator,          samples\_per\_epoch=2000,          nb\_epoch=50,          validation\_data=validation\_generator,          nb\_val\_samples=800)  Example of transforming images and masks together.  # *we create two instances with the same arguments*  data\_gen\_args = **dict**(featurewise\_center=**True**,                       featurewise\_std\_normalization=**True**,                       rotation\_range=90.,                       width\_shift\_range=0.1,                       height\_shift\_range=0.1,                       zoom\_range=0.2)  image\_datagen = **ImageDataGenerator**(\*\*data\_gen\_args)  mask\_datagen = **ImageDataGenerator**(\*\*data\_gen\_args)  # *Provide the same seed and keyword arguments to the fit and flow methods*  seed = 1  **image\_datagen.fit**(images, augment=**True**, seed=seed)  **mask\_datagen.fit**(masks, augment=**True**, seed=seed)  image\_generator = **image\_datagen.flow\_from\_directory**(      'data/images',      class\_mode=**None**,      seed=seed)  mask\_generator = **mask\_datagen.flow\_from\_directory**(      'data/masks',      class\_mode=**None**,      seed=seed)  # *combine generators into one which yields image and masks*  train\_generator = **zip**(image\_generator, mask\_generator)  **model.fit\_generator**(      train\_generator,      samples\_per\_epoch=2000,      nb\_epoch=50) |