

# Computer Vision : Technical Report 4<sup>th</sup> Project

Papadopoulos Aristeidis A.M.:57576

## 1. Non Pre-trained network:

- **Layers that were used:**

- Dense Layer(`tf.keras.layers.Dense(units, activation...)`):

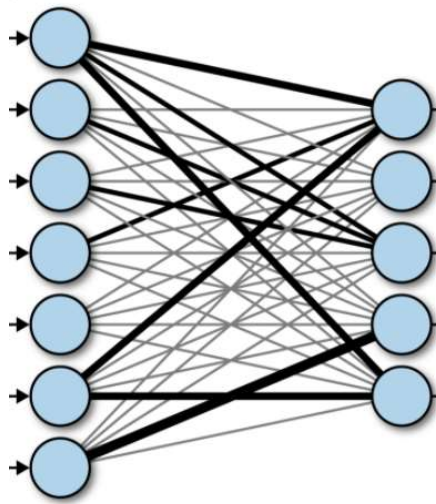


Image 1. Example of a Dense (Fully Connected) Layer.

The simplest kind of layer, each neuron of this layer is connected to every single neuron of the previous layer, so for  $N$  neurons, the connections made are  $2^N$ . Due to this number of connections, much of computational power is needed to calculate and update the weights of each connection. The parameter “units” is the number of neurons that this layer will have, while the parameter “activation” refers to the activation function that will be used. Other parameters are not analyzed here and are used with their default values.

- Flatten Layer(`tf.keras.layers.Flatten()`):

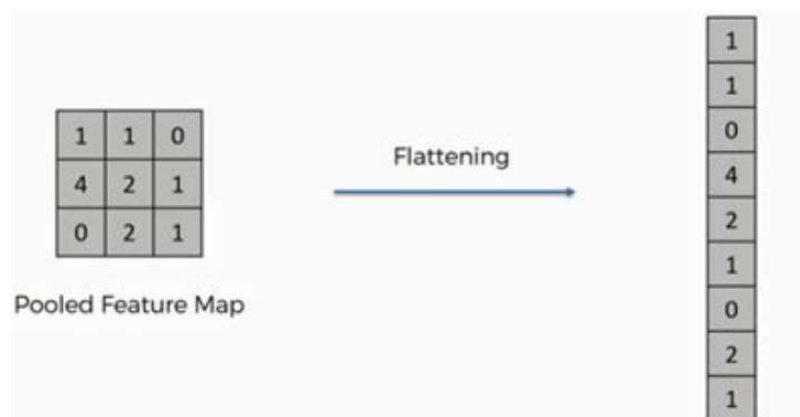


Image 2. Example that depicts the flattening procedure.

This layer converts its input to a vector with length equal to the product of its input dimensions. For example, an image with dimensions 40x40x2 will be converted to a vector with length 3200.

- **Max-Pooling Layer** (`tf.keras.layers.MaxPooling2D(pool_size, strides, padding)`):

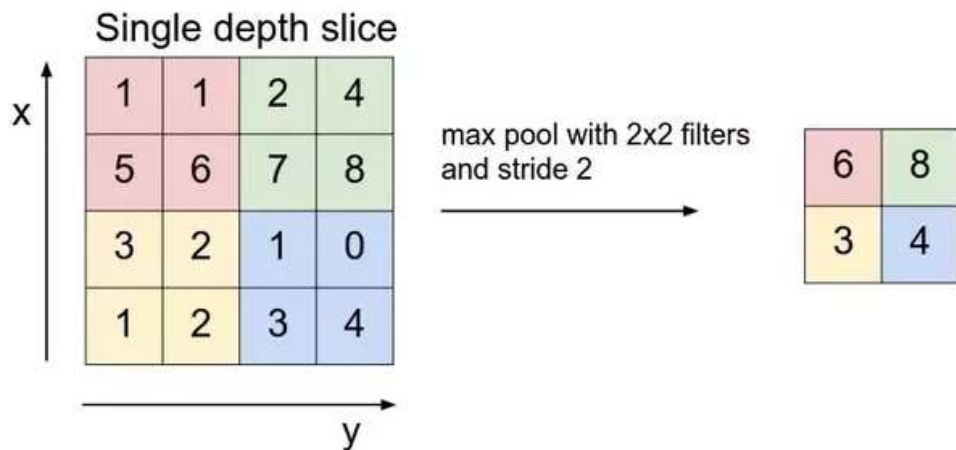


Image 3. Example illustrating the Max-Pooling operation.

Pooling reduces the dimensions of its input, as the input flows throughout the network. The reduction is essentially subsampling and by doing that, a reduction in computational costs is achieved. The reduction percentage is dependent on the dimensions of the filter defined by the user and by how far the window will slide (stride), between two consecutive subsamplings. Usually these layers are used after convolutional layers. Depending on the kind of subsampling, either the maximum value or the median value can be selected. For this project, only max pooling was used.

- **Drop-Out Layer**(`tf.keras.layers.Dropout(rate)`):

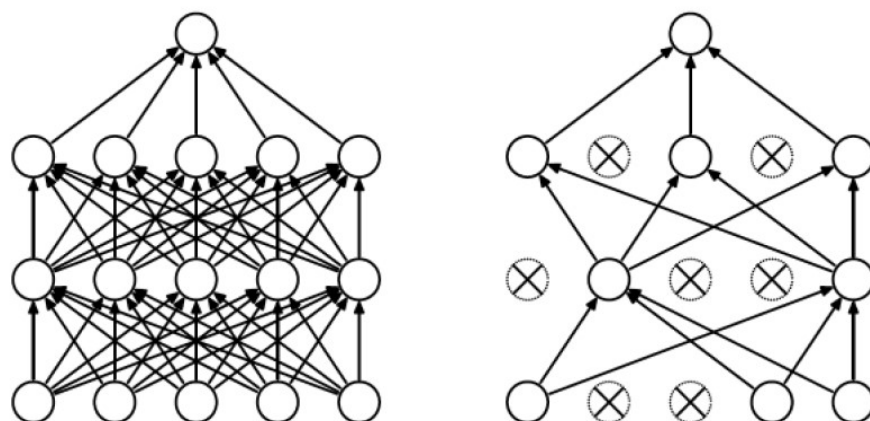


Image 4. Example illustrating the Dropout function.

Dropout minimizes overfitting, disabling neurons for one epoch randomly. The random parameter is used as a probability, ranging from 0 to 1. The larger the probability, the more neurons are disabled.

- `Convolutional(tf.keras.layers.Conv2D(filters, kernel_size, strides, padding, activation...))`:

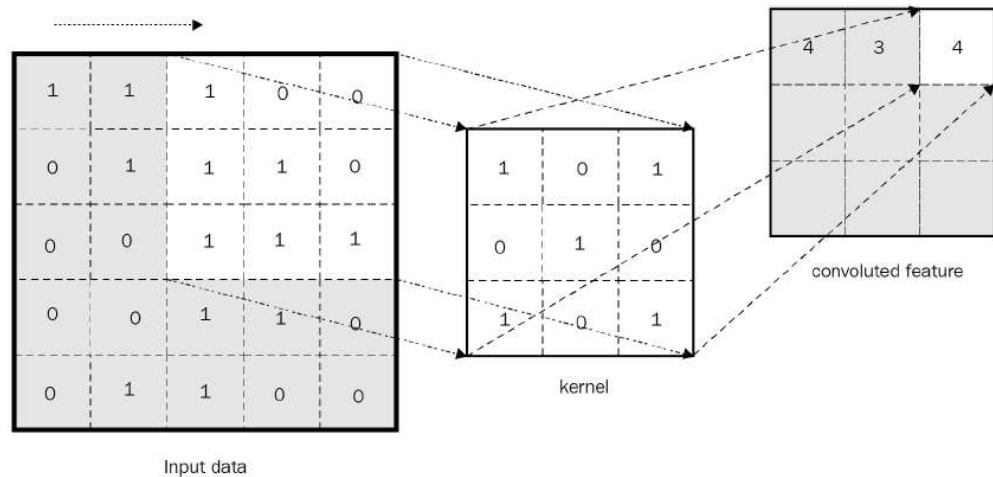


Image 5. Example illustrating the convolution made at Convolutional Layers (kernel size 3x3).

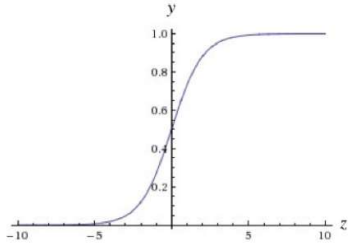
This layer convolves its input, thus reducing the dimensions. The information extracted is locally correlated, while the result of each convolution is a feature that is saved in a feature map. The convolution operation uses a kernel, whose dimensions can vary, extracting different kind of features. The original dimensions can stay the same, as long as padding is used.

- **Hyper-parameters for each layer:**

- **Activation:**

An activation function is used to define the output of a neuron for whatever input it may have. The kind of function influences the output, so for different functions different output will be achieved.

Name	Mathematical Equation	Output-Explanation
Relu(Rectified Linear Unit)	$g(t) = \max(0, t)$	

		Negative inputs are set to zero, while positive values remain the same.
Sigmoid	$g(t) = \frac{1}{1 + e^{-t}}$	 <p>When the input is very low, output is near zero. On the contrary, when the input has a high value, output is set near one.</p>
SoftMax	$g(f_1, \dots, f_c) = \left( \frac{e^{f_1}}{\sum_j e^{f_j}}, \dots, \frac{e^{f_c}}{\sum_j e^{f_j}} \right)$	All the inputs are converted into probabilities.

Also, it is worth mentioning that while there are many activation functions to choose from, almost exclusively the Relu activation function is used, or one of its variants.

➤ **Pool\_size, stride, Padding, Filters, Kernel\_size:**

Pool size defines the size of the window where the subsampling will happen, while stride defines how far the window will shift between two successive subsamplings. If padding is used, the input dimensions remain the same, either by adding zeros or by mirror the last elements of the input. Filters determine the number of convolutions that will happen, while kernel size determines the dimensions of the kernel used in the convolution operation.

- **General Hyper-parameters:**

➤ **Batch\_size:**

Its value determines how many images are going to be passed through the network at one given moment. Ideally, we would like to pass all the training samples together, but due to hardware limitations that is not possible.

➤ **Shuffle, Class\_mode, Rescale:**

Shuffle is a Boolean option whether to read randomly or not the data during training, while class mode is selected as categorical,

due to the fact that the sample classes' are more than two. Finally, rescale normalizes the input values in the range zero to one.

➤ **Target\_size:**

Determines the dimensions of the images to be read. Must be selected in a way that the input is not distorted, because high large values stretch the input, while small values compress it. The larger the values are, the more time is needed to read and pass the input through the network, thus delaying the training process, while not necessarily increasing accuracy. For this project, the target size was selected to be 100x100, as a middle solution between the input shapes of the dataset given and the time needed to train the models developed.

➤ **Validation\_split & seed:**

Due to the absence of a separate validation dataset, I chose to split the given dataset, with a percentage of 25%. This means that images were randomly selected to be placed into the validation set. Seed is used and set to one, in order to not give any kind of weight to the images. Using a validation set allows me to have an idea on the generalization capabilities of the network.

The aforementioned variables are hyper-parameters that define how the input is going to be used and read by the network. On the contrary the following variables define how the process of training will occur.

➤ **Epoch:**

One epoch is defined as the passing of all the available training samples once. Having a high value risks the appearance of overfitting, while a low value creates the risk of underfitting, i.e., the inability of the network to classify the training samples.

➤ **Steps\_per\_epoch:**

Steps per epoch determine the number of batches per epoch. Having a small value means that few batches will be used per epoch, while the max value must be equal to the product of the batch size and the steps per epoch, so as to not surpass the number of images available. Its value is set automatically to its default.

Furthermore, callbacks are used to optimize training. With callbacks, two functions are utilized. Firstly, validation loss is tracked and the weights that minimize it are saved. Secondly, a check is made whether validation loss has improved or not within a specific number of epochs, if it has not, training stops and the weights that minimize validation loss are restored. Using callbacks, the training process is sped up and overfitting can be avoided.

Finally, an optimizer to enforce the back-propagation algorithm is used, with learning rate equal to 0.001, while the loss function is defined as cross-entropy, since more than two classes are present in the dataset. The optimizers used are Adam, Adadelta & Adagrad.

- **Network architectures:**

Two different architectures were implemented for this project, one based solely on convolutional layers and one typical architecture containing convolutional and fully connected layers. Even though batch normalization was tried, it delayed training so ultimately it wasn't used.

➤ **Network based solely on Convolutional Layers:**

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(256, (5, 5), strides=(2, 2), activation='relu', input_shape=(100,100, 1), padding = 'same'),
    tf.keras.layers.Conv2D(256, (4, 4), strides=(2, 2), activation='relu', padding = 'same'),
    tf.keras.layers.Conv2D(128, (3, 3), strides=(1, 1), activation='relu', padding = 'same'),
    tf.keras.layers.Conv2D(128, (3, 3), strides=(1, 1), activation='relu', padding = 'same'),
    tf.keras.layers.MaxPooling2D(pool_size = (2, 2), strides=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(rate = 0.3),
    tf.keras.layers.Dense(34, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
model.summary()
```

Image 8. 1<sup>st</sup> network architecture, with indicative values for the hyperparameters and the optimizer.

Four convolutional layers were used, to extract sufficient amount of features and to provide the depth needed to the network. Then, a Max-Pooling layer was used to reduce dimensions and computational costs of the model, a flattening layer to convert all the feature maps into one vector, a Drop-out layer to minimize the effect of over-fitting and one dense layer to act as a classifier, where the neurons are equal to the number of classes in the given dataset (34) and with softmax activation function, so the model output is in probability format.



## ➤ A full CNN network architecture:

```
#Complete CNN
model1 = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(256, (4, 4), strides=(2, 2), activation='relu', input_shape=(100,100, 1), padding = 'same'),
    tf.keras.layers.Conv2D(256, (3, 3), strides=(2, 2), activation='relu', padding = 'same'),
    tf.keras.layers.Conv2D(128, (3, 3), strides=(2, 2), activation='relu', padding = 'same'),
    tf.keras.layers.Conv2D(64,(2,2),strides=(1,1),activation='relu',padding='same'),
    tf.keras.layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2)),
    tf.keras.layers.Dropout(rate = 0.3),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(rate = 0.25),
    tf.keras.layers.Dense(34, activation='softmax')
])
model1.compile(optimizer=tf.keras.optimizers.Adam(),#or adamax
               loss = 'categorical_crossentropy',
               metrics = ['accuracy'])
```

Image 8. 2<sup>nd</sup> network architecture, with indicative values for the hyperparameters and the optimizer.

In this network, several convolutional layers are used in the beginning for the same reasons as before, followed by a Max-Pooling layer, a Dropout and a Flattening Layer. Then, a few fully connected layers with a Dropout layer to simplify the final classification. The neurons in the fully connected layers are powers of two and with values not too small or not too high as to not overfit the network. Relu was set as activation function, because after some research it yielded the best results, while padding is used to avoid problems with image dimensions as the images flow through the network.

### • Hyper-parameter values – Results comparison

Batch size was set to the highest value possible, so as not cause OOM error (Out of Memory), due to the big size of the images. The dimensions with which to read the images is an important factor, because it is possible by magnifying or shrinking the images, to change each image's features. The conclusions extracted are not definite, due to randomization of the training.

Training Batch Size	Validation Batch Size	Testing Batch Size	Image dimensions
800	700	1000	(100,100)

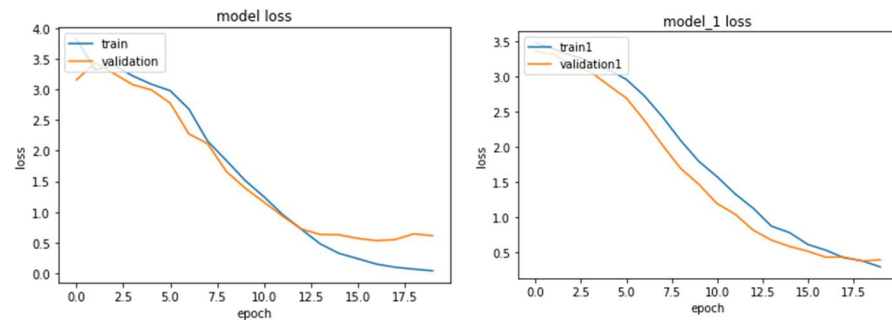
## ➤ Epoch value effect:

Epochs must be set to a value that does not cause the network to overfit, nor to cause the network to underfit. Overfitting causes the network to classify accurately only the training samples, while underfitting cause the network to classify wrongly even the training samples. A large epoch number also means a lot of

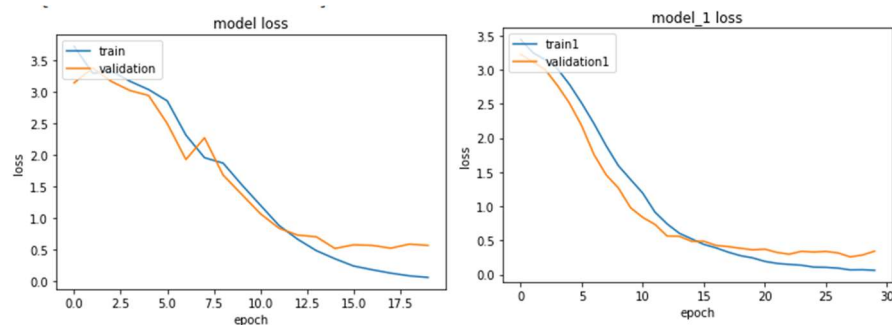
computational time in training. Having those things in mind, epoch number was set to 20, 30 40 and 60. The results of each trial in the test set can be seen in the following table.

Epochs	1st network accuracy	2nd network accuracy	1st network loss	2nd network loss
20	0.9195	0.8986	0.4294	0.3895
30	0.9204	0.9558	0.3959	0.2197
40	0.9269	0.9521	0.3269	0.2496
60	0.9139	0.9395	0.3687	0.2746

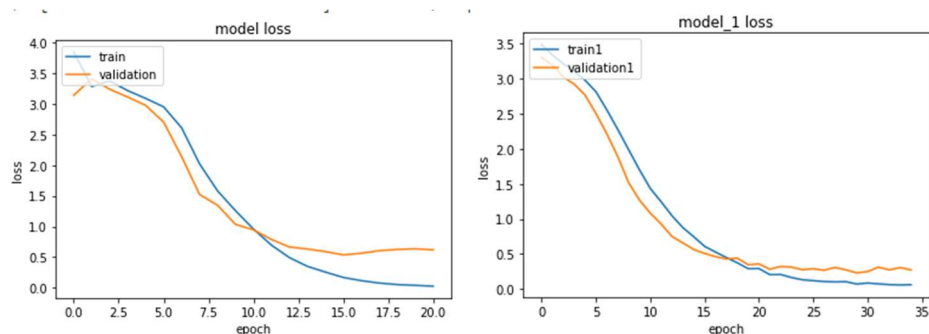
It can be observed that accuracy increases and then decreases, so probably overfitting starts to occur, due to the high number of epochs. Also, based on the following diagrams, because of the use of callbacks, the max epoch number is not even reached, in the case of 40 and 60 epochs, while validation loss begins to fluctuate, a sign of overfitting. So, 30 epochs seem to be the optimal number.



Images 9,10. Decrease of loss during training for the two models for 20 epochs (1<sup>st</sup> model on the left, 2<sup>nd</sup> on the right).

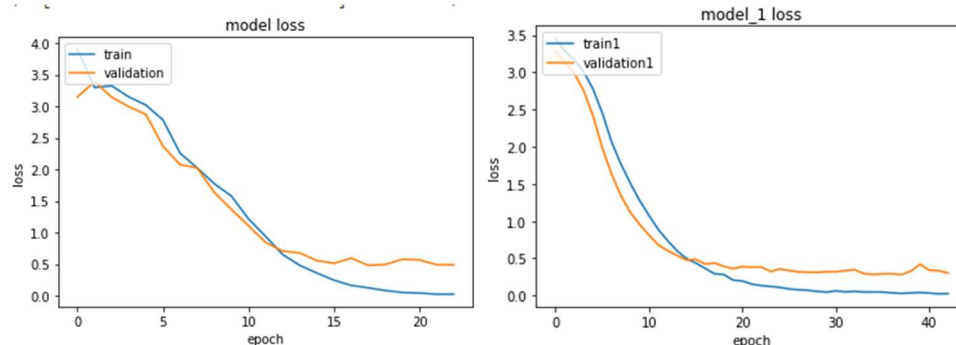


Images 11,12. Decrease of loss during training for the two models for 30 epochs (1<sup>st</sup> model on the left, 2<sup>nd</sup> on the right).



Images 13,14. Decrease of loss during training for the two models for 40 epochs (1<sup>st</sup> model on the left, 2<sup>nd</sup> on the right).





Images 15,16. Decrease of loss during training for the two models for 60 epochs (1<sup>st</sup> model on the left, 2<sup>nd</sup> on the right).

### ➤ Callbacks effect:

Callbacks	Accuracy 1 <sup>ης</sup> αρχιτεκτονικής	Accuracy 2 <sup>ης</sup> αρχιτεκτονικής
Με callbacks	0.9204	0.9558
Χωρίς callbacks	0.9037	0.9376

If callbacks are not used, training goes on regardless of whether the validation loss is improving or not. So, overfitting danger lurks as the weights drift from their optimal values. As expected, the use of callbacks improves network training.

### ➤ Number of neurons effect:

The selection of neurons in each layer is random. Of course, the higher the neurons are, the more time is needed for training since more parameters must be trained. Also, since it is not essential for the accuracy to increase with more neurons, there is the chance that the network will overfit.

Layer (type)	Output Shape	Param #
conv2d_183 (Conv2D)	(None, 50, 50, 256)	6656
conv2d_184 (Conv2D)	(None, 25, 25, 256)	1048832
conv2d_185 (Conv2D)	(None, 25, 25, 128)	295040
conv2d_186 (Conv2D)	(None, 25, 25, 128)	147584
max_pooling2d_45 (MaxPooling)	(None, 12, 12, 128)	0
flatten_44 (Flatten)	(None, 18432)	0
dropout_49 (Dropout)	(None, 18432)	0
dense_92 (Dense)	(None, 34)	626722
Total params: 2,124,834 Trainable params: 2,124,834 Non-trainable params: 0		
conv2d_175 (Conv2D)	(None, 50, 50, 256)	4352
conv2d_176 (Conv2D)	(None, 25, 25, 256)	590080
conv2d_177 (Conv2D)	(None, 13, 13, 128)	295040
conv2d_178 (Conv2D)	(None, 13, 13, 64)	32832
max_pooling2d_43 (MaxPooling)	(None, 6, 6, 64)	0
dropout_46 (Dropout)	(None, 6, 6, 64)	0
flatten_42 (Flatten)	(None, 2304)	0
dense_88 (Dense)	(None, 256)	590080
dense_89 (Dense)	(None, 128)	32896
dropout_47 (Dropout)	(None, 128)	0
dense_90 (Dense)	(None, 34)	4386
Total params: 1,549,666 Trainable params: 1,549,666 Non-trainable params: 0		

Image 17,18 Trainable parameters after each layer and their total (1<sup>st</sup> network on the left, 2<sup>nd</sup> on the right).

➤ **Kernel size, stride & pool size** effect:

Kernels and strides affect the part of the image that is going to be the input of the neurons. Especially in convolutional layers, the size of the kernels affects the neurons' output, as the kernels convolve with the input. So, for high kernel dimensions, the convolution will be larger, while with high stride dimensions, the pixels omitted will be more but at the same time training time will be sped up. The possibility arises that key features are going to be omitted. Finally, large pool size means that more pixels will be rejected, so again key features might be omitted.

Kernel Size	1 <sup>st</sup> network accuracy	2 <sup>nd</sup> network accuracy
(3,3)	0.919	0.9432
(5,5)	0.9246	0.92
Strides Size	1 <sup>st</sup> network accuracy	2 <sup>nd</sup> network accuracy
(2,2)	0.8781	0.9507
(3,3)	0.5537	0.6198

I concluded that the optimal solution is the use of different kernel sizes and strides and not one size for all of them.

➤ **Optimizer** effect:

The main difference between optimizers is the way they calculate and change the learning rate, the minimization of loss and the calculation of each neurons' weights. Some optimizers are:

- Adagrad: Each element is divided by the square root of per element summation of the gradient's squares.
- Adadelata: Each sum is replaced by an ever-smaller sum, resulting from the change in gradients.
- Adam: The most commonly used optimizer, it's a combination of the above.

The proper selection of optimizer is an important factor, because each optimizer affects differently the learning rate, and thus the weights of each neuron.

Optimizer kind	1 <sup>st</sup> network accuracy	2 <sup>nd</sup> network accuracy
Adam	0.9204	0.9265
Adadelata	0.1661	0.0279
Adagrad	0.2457	0.0935

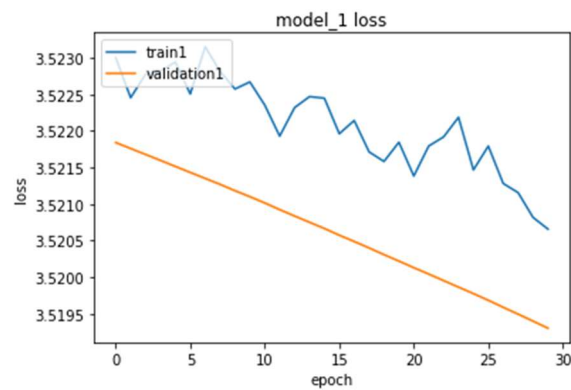
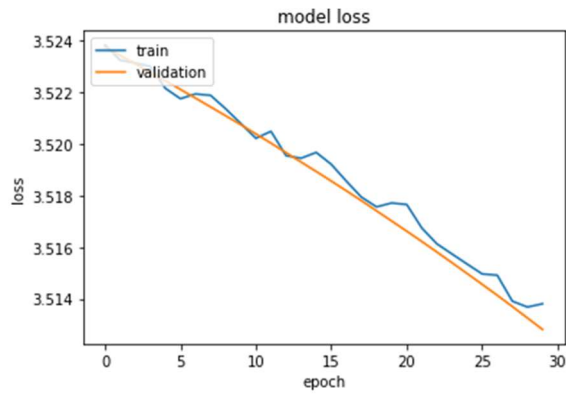


Image 19,20. Validation and training loss fluctuation during training using Adadelta optimizer (1<sup>st</sup> architecture on the upper side, 2<sup>nd</sup> on the lower side).

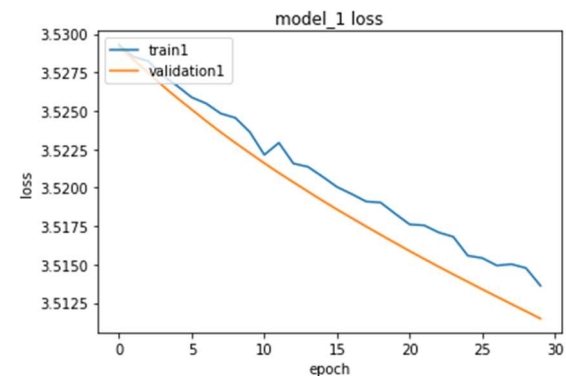
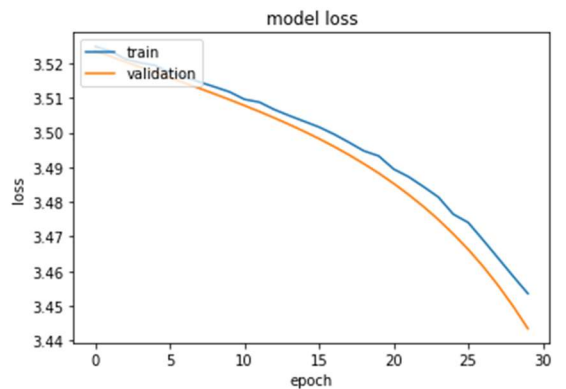


Image 21,22. Validation and training loss fluctuation during training using Adagrad optimizer (1<sup>st</sup> architecture on the upper side, 2<sup>nd</sup> on the lower side).

Based on the results and on the diagrams above, it can be observed that the losses, instead of decreasing abruptly, they are decreasing linearly, probably due to the way the losses and learning rate are calculated. Furthermore, it is possible that with a

higher epoch number, the results could be better. In conclusion, the optimal choice seems to be Adam, since less time is needed to produce the required results.

## 2. Pre-trained network:

The two architectures that I chose were Xception and InceptionResNetV2 and the reason why they were selected is the relatively small size, with 88MB and 215MB respectively.

- **1<sup>st</sup> architecture: Xception**

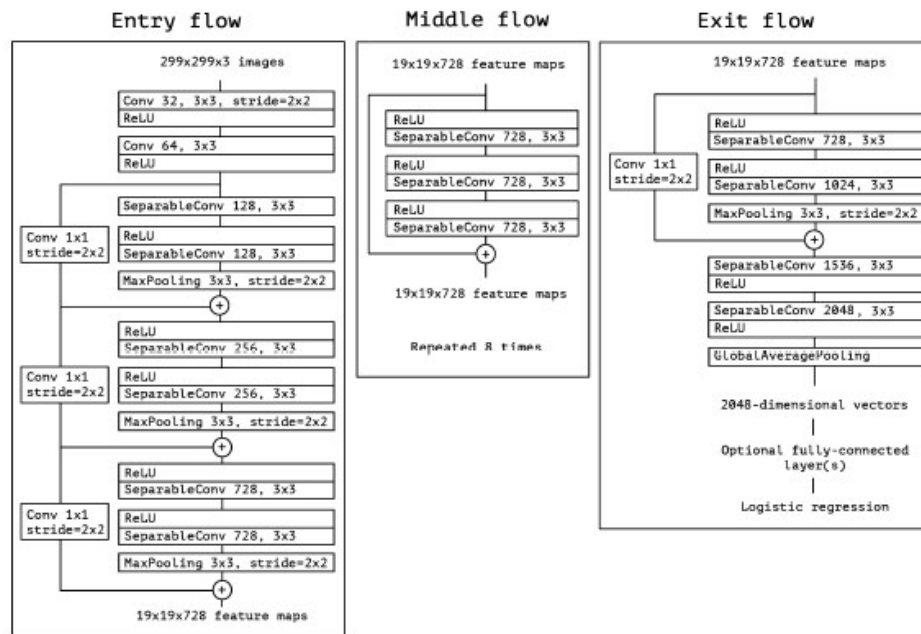


Image 22. Basic Xception model.

```
model.add(layers.Dense(32,activation='relu'))
model.add(layers.Dropout(rate = 0.4))
model.add(layers.Flatten())
model.add(layers.Dropout(rate = 0.3))
model.add(layers.Dense(34, activation='softmax'))

model.summary()
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.Adam(),
              metrics=['acc'])
```

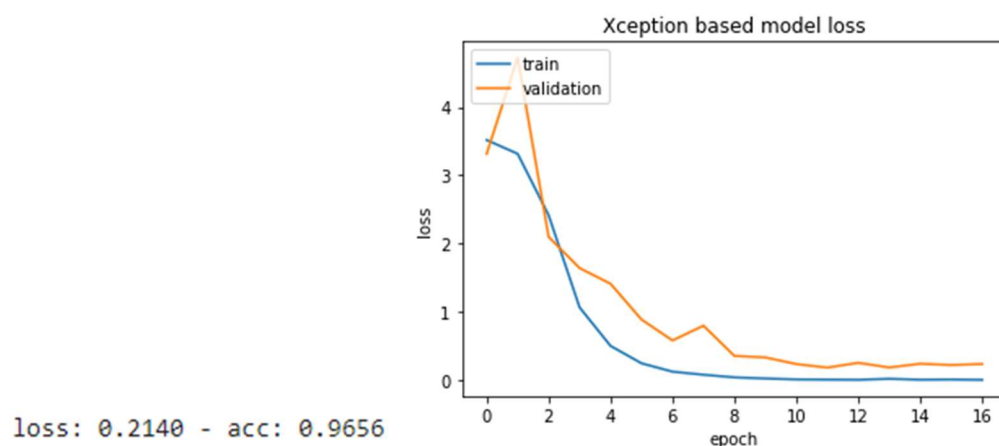
Image 23. Changes made to Xception model.

The basis of the network is the use of convolutions, with layers implementing vanilla convolution, separable and Depthwise convolution. 36 Convolutional layers are used to produce the feature maps, where between some convolutional layers Max-Pooling layers are placed. To increase the network's accuracy, I added a small dense layer with 32 neurons and with Relu activation function, followed by

dropout layer with probability 0.4. Then, a flatten layer was added, followed by a dropout layer with 0.3 probability and finally the fully connected layer that classifies the inputs. Lastly, Adam was set as to be the optimizer.

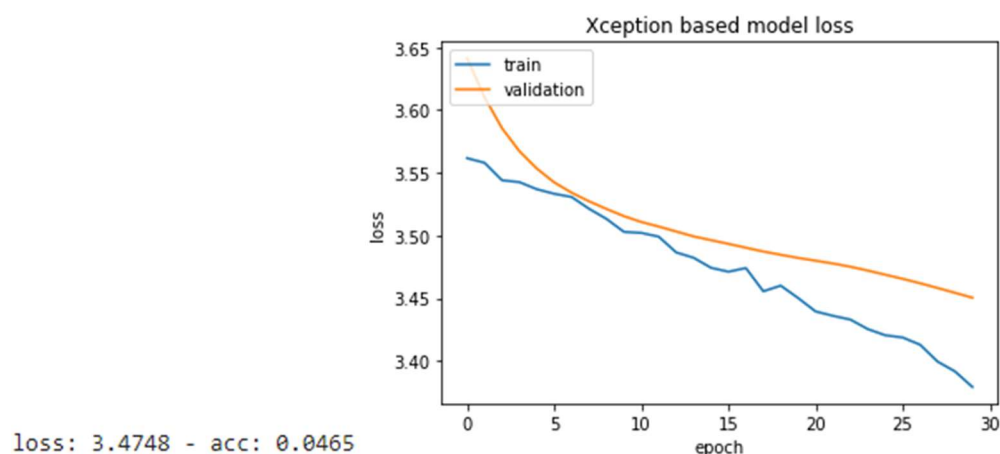
### ➤ Network Hyper-parameters:

Having as a fact that there are millions of neurons, 20 epochs were set, as to speed up the training. Due to the large size of the network, smaller batch size was used, while the images were read with dimensions 100x100. Callback are used to avoid overfitting, with 5 epochs patience.



Images 24,25. Accuracy of the final model and the fluctuation of validation loss during training.

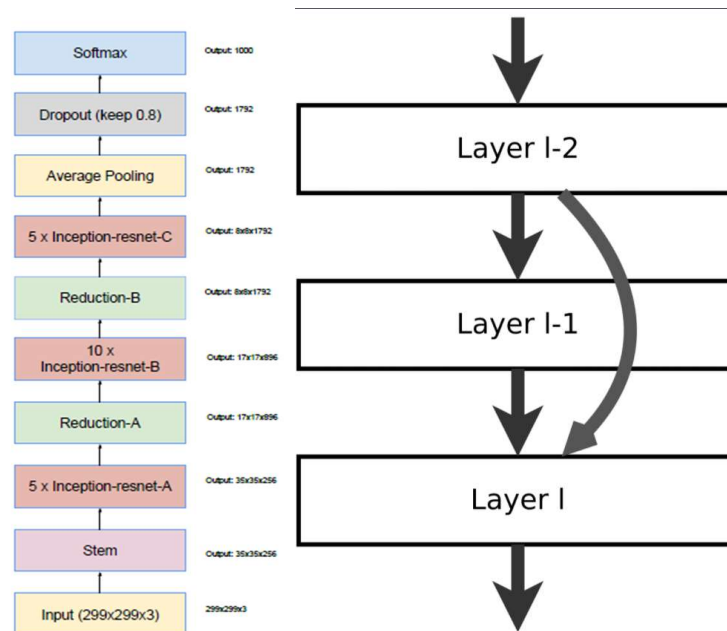
An overshoot can be observed near the beginning of training, which probably is due to the fact that the weights are not randomized, but are the saved weights from the Imagenet problem, so some epochs are needed for the weights to adjust to the current problem. Due to callbacks, training stops earlier and finally, compared to previous architectures, accuracy is increased by 2%. The results changing different network parameters such as optimizer, epochs etc can be seen in the following images.



Images 26,27. Results using different values for the different parameters of the network.

Based on the above indicative results, the changes made did not improve accuracy, instead accuracy worsened, as can be seen by the accuracy and the loss during training.

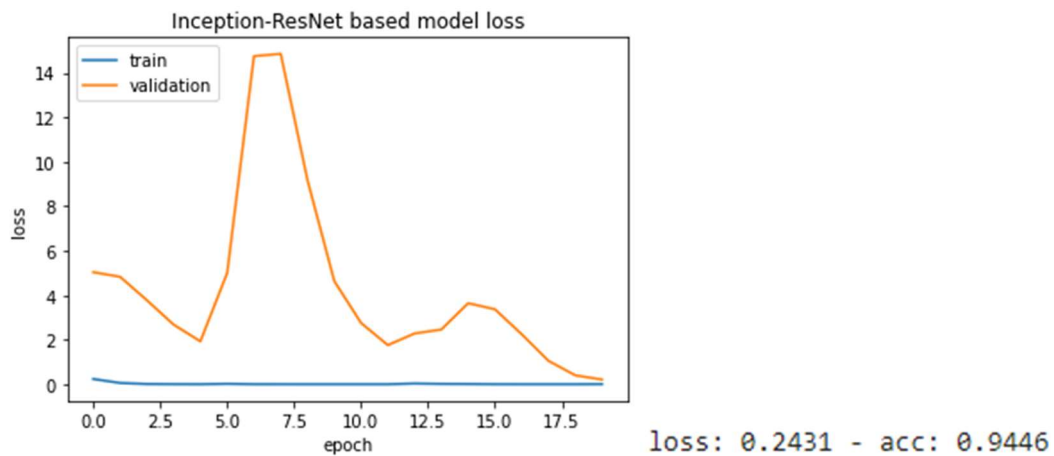
- **2<sup>nd</sup> architecture: InceptionResNetV2**



Images 28,29. Basic InceptionResNetV2 network topology (left), with a Resnet connection example (right).

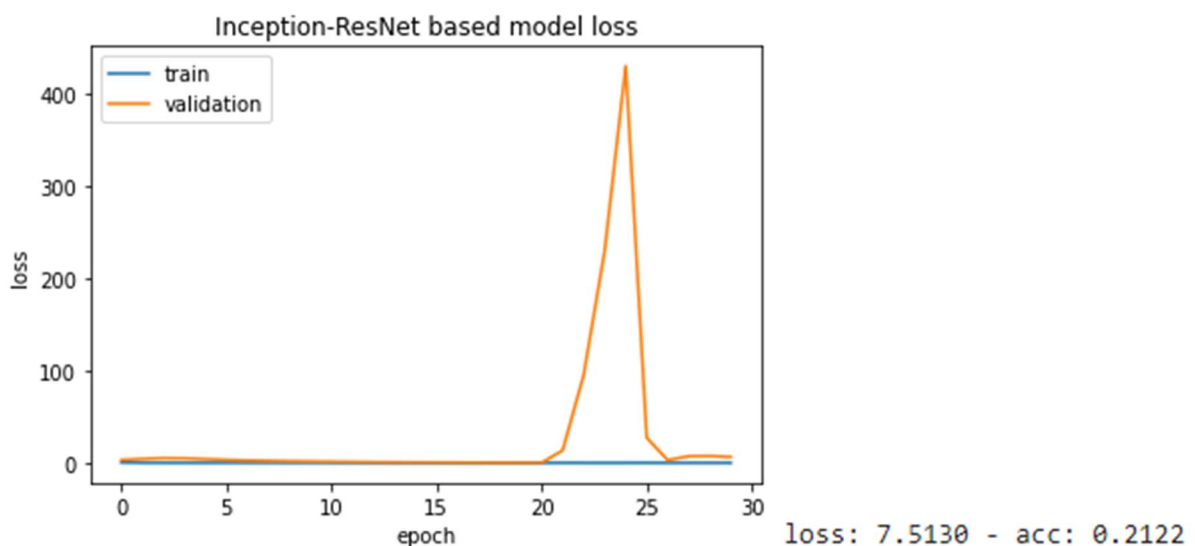
A Residual Network(Resnet) is a networks whose layers due to shortcuts and jumps, have skipped connections in a way that the layers are not linearly connected anymore. Since some connections are skipped, training time can be improved. Every part of the architecture contains a mix of Convolutional Layers with Max-Pooling layers. Those parts are not analyzed in depth, as this is not the requested in the current project. I inserted a dropout layer to simplify the existing architecture of 55.000.000 neurons, a flatten layer, followed by a dropout layer and finally the fully connected layer to classify the images used. The same hyper-parameter tuning with Xception was made, with the results following.





Images 30,31. Results from the use of InceptionResNetV2 model.

Based on the above, overshoots of validation loss can be seen, though in the end an acceptable accuracy is achieved, a little less though than the Xception one. Changing some parameters such as the number of epochs (from 20 to 30) and batch size, resulted in lower accuracy than before, probably due to overfitting.



Images 32,33. Results for using different values at the different parameters of the network.

## • References

- **Richard Szeliski**: Computer Vision Algorithms and Applications
- Keras Documentation: <https://keras.io/api/>
- Wikipedia : <https://en.wikipedia.org/>
- **Arcangelo Distanto, Cosimo Distanto**: Handbook of Image Processing and Computer Vision, Volume 3: From Pattern to Object
- **Nikhil Buduma, Nicholas Locascio** : Fundamentals of Deep Learning, Designing Next-Generation Machine Intelligence Algorithms
- **Yoshua Bengio, Aaron Courville, Ian Goodfellow** - Deep learning
- **Francois Chollet**: Xception: Deep Learning with Depthwise Separable Convolutions
- **Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi**: Inception-v4, Inception-Resnet and the Impact of Residual Connections on Learning