Machine Learning 3 – Practice I
Convolutional Neural Nets

MBD – April 2020 intake

Team B

Timo Bachman

Alberto De Roni

William Kingwill

Qing Loh

Francisco Mansilla

Umut Varol

Julius Von Selchow

1 – Cats & Dogs Image Classification

In 01_cnn_template.ipynb, we performed steps 1,2 and 3: added a CNN, flattened the last layer and added a dense layer.

2 – Regularization

In 02_cnn_template.ipynb (overfitting scenario), data augmentation was performed with *ImageDataGenerator*. Furthermore, we will explain the parameters of this function and what dropout regularization are all about.

2.1 Data Augmentation

The basic idea behind the ImageDataGenerator class is to generate "new" training samples on the fly in order to introduce more generalizability in a neural network. Mind how only the new, randomly generated data is used in the training process (the network never sees the original pictures), and how the process is done at training time as a direct preprocessing step and not prior to it. The class is used in the following way:

- Present original input batch to the ImageDataGenerator
- The class transforms batch images with a series of transformations, rotations, horizontal (and occasionally vertical) flips and other image modifications
- The new randomly generated batch is returned to the calling function to be used for training

The class manages to improve the network's ability to generalize by making it focus on the more robust features of an image. This is achieved by the class's non-additive nature, meaning that the original data is not shown to the model. If this wasn't the case, the class's purpose would be defeated as the model would see the original picture at each iteration. By only being shown slightly modified versions of the original image, the model learns to generalize.

One last very important detail is that this is only applied to a sample of training data and not validation nor test data.

Some selected parameters of the *ImageDataGenerator* class are elaborated below.

● *rotation_range:* input a value between 0 and 180 which represent a degree range within which the picture will be randomly rotated - **an input of 40 would result in a random rotation between 0 and 40 degrees**

● *width_shift_range:* input a value between 0 and 1 to be the range within which the picture will be randomly translated on the horizontal axis. If >=1, the value represents the number of pixels - **an input of 0.2 would imply a 20% range of the total picture's horizontal axis within which the new image may be randomly translated anywhere**

● *height_shift_change:* input a value between 0 and 1 to be the range within which the picture will be randomly translated on the vertical axis. If >=1, the value represents the number of pixels - **an input of .2 would imply a 20% range of the total picture's vertical axis within which the new image may be randomly translated anywhere**

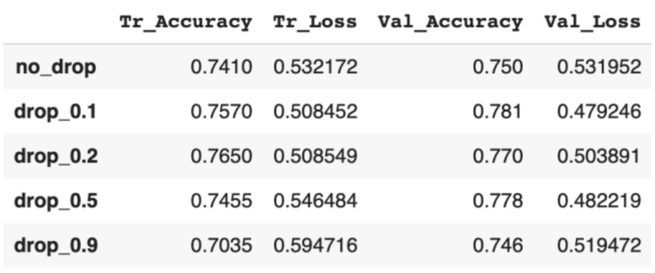
- *shear_range:* input a float to determine shear intensity, meaning the shear angle in counter-clockwise direction in degrees - **an input of 0.2 would translate to a shear angle of 0.2 degrees in counterclockwise direction**
- *zoom_range:* a range for random zooming, can take [lower, upper] or a float as input - **an input of 0.2 would result in a [0.8, 1.2] range**
- *horizontal_flip:* T/F to determine whether or not to randomly flip the input picture on the horizontal axis - **True would result in a horizontally flipped version of the original input**
- *fill_mode:* This is the strategy that is used for filling in the pixels created by the ImageDataGenerator, these can appear after a rotation or within a width/height shift. In other words, the points outside the input boundaries are filled according to one of the following modes:
    - constant: kkkkk|abcd|kkkkk (caval=k)
    - nearest: aaaaaa|abcd|ddddddd
    - reflect: abcddcba|abcd|dcbaabcd
    - wrap: abcdabcd|abcd|abcdabcd
    - **setting 'nearest' would result in the newly created pixels being copies of their nearest original**

2.2 Dropout

Dropout is considered to be one of the key advances in Deep Learning. With a probability p, we take a node and remove it from the dropout layer in the neural network for a single gradient step or iteration, randomly for each step. The probability p is encoded as dropout rate, taking a value between 0 and 1. A value of 0 implies no dropout regularization at all and would return an identical model as if there was no dropout layer. Conversely, a value of 1 will force to drop out every node, leading to an overly simple model that will not learn at all. With a probability *p* equal to the specified drop rate, the free parameters (weights) are set to 0, while the remaining inputs are scaled up to *1/(1-p)* so that the sum over all inputs is constant.

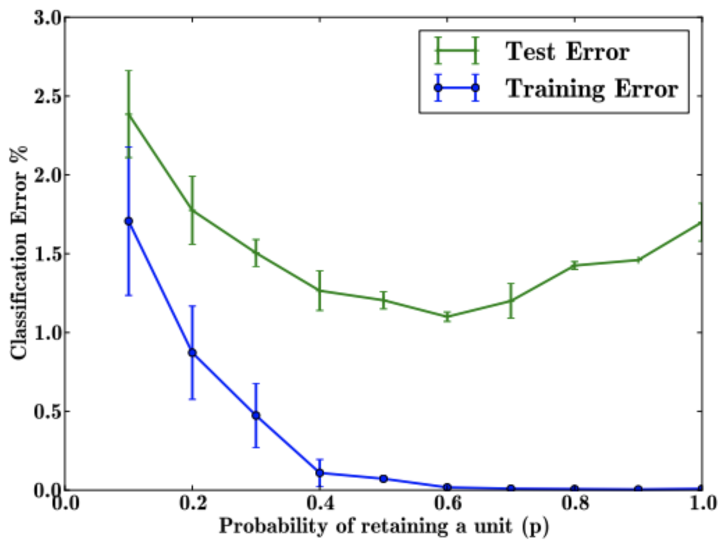
Dropout is a powerful regularization technique for neural networks as each iteration is forced to be independently useful and cannot rely on the output of other nodes.

Finally, comparing different levels of the dropout rate, we can observe a general decrease in training accuracy for increasing dropout values. Similarly, the validation accuracy followed this pattern, but in a weaker way. What we would expect here is a weak regularization and higher tendency to overfit with dropout rate of 0.1. This should result in a higher training accuracy score at the cost of lower performance on the validation set. By the same token, a dropout rate of 0.9 would imply strong regularization, i.e. less overfitting and a more generalized model. Such dropout rate should yield a relatively higher validation accuracy while performing worse on the training set.



	Tr_Accuracy	Tr_Loss	Val_Accuracy	Val_Loss
no_drop	0.7410	0.532172	0.750	0.531952
drop_0.1	0.7570	0.508452	0.781	0.479246
drop_0.2	0.7650	0.508549	0.770	0.503891
drop_0.5	0.7455	0.546484	0.778	0.482219
drop_0.9	0.7035	0.594716	0.746	0.519472

We explain our observations with the findings of Srivastave et al. (2014) that understood the relationship of increasing values for the dropout rate (1 - probability of retaining a unit) as in the below graph:

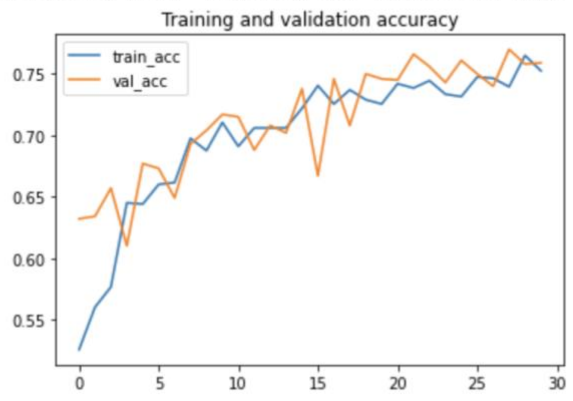


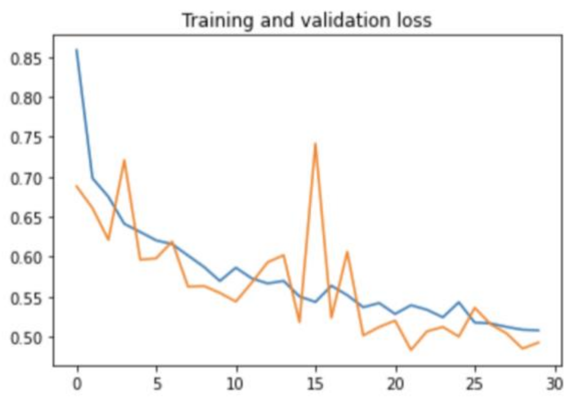
With the number of hidden units in the dropout layer adjusting after the dropout, the training error has been found to increase for higher dropout rates (shrinking probability of retaining a unit), i.e. reducing the training accuracy. The dropout layer decreases the test error by increasing the generalizability of the model. However, there is a tradeoff between dropping nodes - thereby increasing the generalizability - and dropping too many nodes and losing important information needed for classification. This is reflected in the graph as the test error increases with high dropout rates above 0.6, i.e. probabilities of retaining a unit below 0.4. As a tradeoff to gain such generalizability, one sacrifices training accuracy.

2.3 Final Model Fit

Finally, in the Python notebook, we fit the final model with the data augmentation in place for training and added a dropout layer to strengthen the robustness of our Convolutional Neural Network. Setting the dropout rate to 0.2, we visualized the training process in the below graphs.

```
Text(0.5, 1.0, 'Training and validation loss')
```





3 – Stand on the Shoulders of Giants – Inception v3

In notebook “03_cnn.ipynb” and “04_cnn_template.ipynb”, the pre-trained net *Inception V3* was implemented. Thereby, we discuss several scenarios.

3.1 Topology & Database

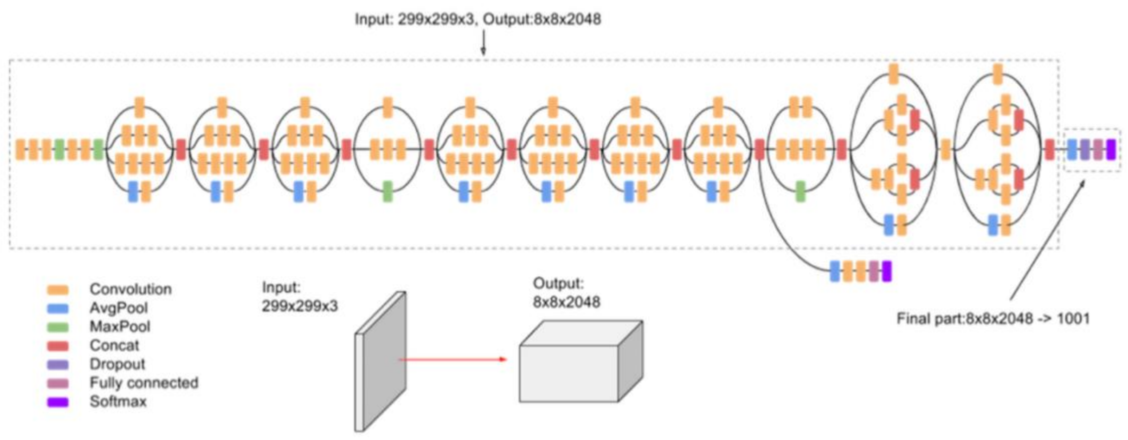
Inception V3 is a CNN created by Google and got its start as a module for Googlenet. It was designed for the classification of objects in Computer Vision. The model consists of 42 layers made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concats, dropouts, and fully connected layers, as seen in the figure below. A noteworthy difference in the architecture of Inception V3 when compared to earlier models is that RMSProp is used as an optimizer and *Batchnorm* is used in the auxiliary classifiers. Moreover, Label Smoothing was introduced, this is a type of regularizing component added to the loss formula that prevents the network from becoming too confident about a class and prevents overfitting.

The model was trained using data from the ImageNet dataset, which has over ten million URLs of labeled images and 22 000 classes. About a million of the images also have bounding boxes specifying a more precise location for the labeled objects, which helps with training.

For the training process of this model, the dataset came from Imagenet and was composed of 1,331,167 images, which were split into training and evaluation datasets containing 1,281,167 and 50,000 images, respectively.

The training and evaluation datasets are kept separate intentionally. Only images from the training dataset are used to train the model and only images from the evaluation dataset are used to evaluate model accuracy.

A result of training on such a huge dataset is that the model is optimized and excellent at feature extraction of images. This is very beneficial as it can be used and applied to any image dataset for feature extraction for CNNs.



3.2 Change of Input Sizes

This is possible because the model is fully convolutional. Convolutions do not account for the image size, they're "sliding filters". If we have big images, we will have bigger outputs, if small images, then small outputs. (The filters, though, have a fixed size defined by kernel_size and input and output filters)

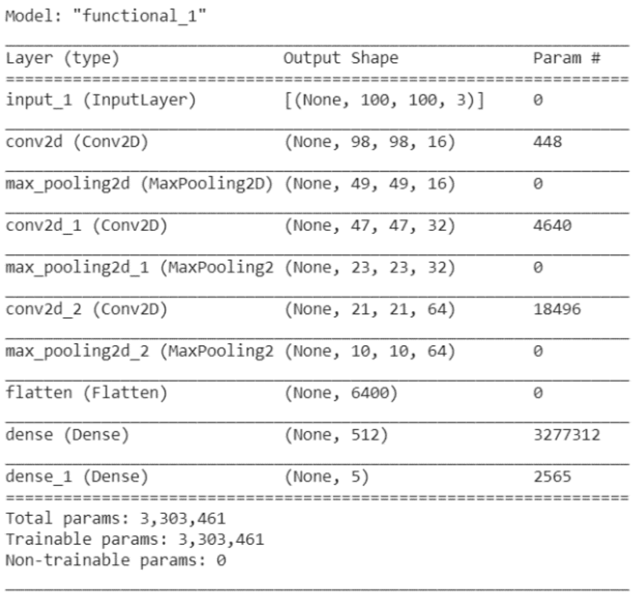
In case that include_top = False, then we need to specify the input shape. From the Keras documentation for inceptionV3, we get the following parameters:

include_top	whether to include the fully-connected layer at the top of the network.
input_shape	optional shape list, only to be specified if include_top is FALSE (otherwise the input shape has to be (299, 299, 3). It should have exactly 3 inputs channels, and width and height should be no smaller than 75. E.g. (150, 150, 3) would be one valid value.

3.3 Change of Database

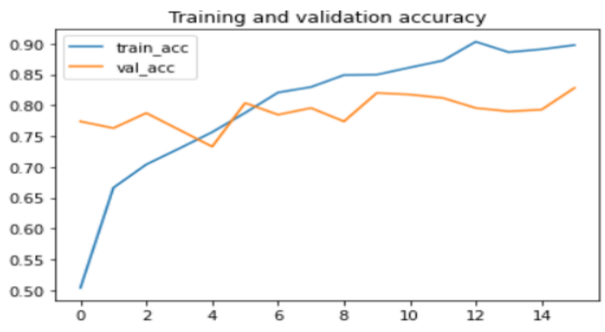
To investigate whether Inception V3 would run well with different objects, it was run on a dataset containing images of flowers.

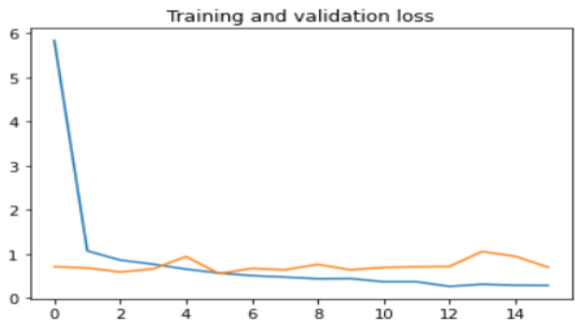
First a small convnet was created to compare the results of the Inception model too. The convnet consisted of 10 layers as seen in the image below. As expected for such a simple model it achieved a very low accuracy on the validation set of 0.35.



```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 100, 100, 3)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 98, 98, 16)        448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 49, 49, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 47, 47, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 23, 23, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 21, 21, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 10, 10, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 6400)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3277312
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 2565
=================================================================
Total params: 3,303,461
Trainable params: 3,303,461
Non-trainable params: 0
_________________________________________________________________
```

The Inception V3 model was then used on the flower dataset as shown in the notebook. A massive improvement was seen with the model being able to achieve an accuracy of 0.8283 on the validation set. The training and validation accuracy and loss is shown in the graph below.





In order to attempt to improve this accuracy further, some fine tuning was done on the InceptionV3 model. The last layers from the 'mixed6' layer weights were unfrozen. Meaning that these weights could be trained to this specific dataset. However, after doing so, a slightly lower validation accuracy was achieved.

4 – Object Detection with YOLO Nets

In this last task, we investigated the highly popular object detection model You-Only-Look-Once (YOLO). Firstly, the original model is introduced, upon which its recent advancements are demonstrated. While YOLO v5 is already online, the academic paper for it has not been published yet. Therefore, the most recent academically documented version is discussed. Later, the YOLO v5 approach was applied to the Roboflow dataset.

4.1 YOLO original

YOLO nets facilitate object detection as a regression problem to spatially separated bounding boxes and directly perform a classification. In one evaluation process, a single neural net predicts such bounding boxes and class probabilities, which allows for direct optimization of detection performance. This unified model allows for quick algorithm runtime that fosters use in real-time application. Furthermore, it excels at encoding contextual information about classes as it reasons from a global point of view. In comparison to historic approaches, this model does not apply a sliding window to detect objects but rather looks at the image as a whole.

To do this, the YOLO system divides the input image into an *S x S* grid. When the center of an object falls into a specific grid cell. That cell is responsible for detecting that object. Each grid cell predicts *B* the bounding box, as well as the *C* conditional class probability. In the prediction for *B* the bounding box also includes a confidence score for those boxes. The confidence score reflects how confident a model is that the box contains an object, as well how accurate it thinks the box that predicts is. If no object exists in that cell, the confidence score would be zero.

Each cell also predicts the *C* conditional class probabilities for that cell. Only one set of class probabilities is being predicted for each cell. At test time, the conditional class probabilities are multiplied by the individual box confidence predictions giving a class-specific confidence score for each box. This score includes both the probability of that class appearing in that box as well as how well the predicted box fits the object.

Ultimately, the model boasts with high generalizability from the representations learned. To connect to previous techniques from this practice, YOLO neural nets make use of dropout and data augmentation to reduce overfitting.

The only apparent downside versus competing models is that the YOLO approach is struggling with an increased localization error, partly offset by the much lower false detection rate on empty spaces. This localization problem goes back to a tradeoff between classification and localization error, where both are equally considered. Treating errors the same in both large and small bounding boxes, the model is prone to incorrect localizations. This is solved by including parameters Ÿ$_{coord}$ and Ÿ$_{noob}$ which increase the loss from bounding box coordinate predictions and decrease the loss from confidence predictions for boxes that don't contain object.

4.2 YOLO v4

This second most recent version of the YOLO neural network shows strong improvements in accuracy and speed of the detection algorithm. By virtue of several modern optimization techniques split into "bag of freebies" and "bag of specials", the convolutional neural net could be upgraded by a considerable amount. The former includes data augmentation strategies such as geometric and photometric distortion. In particular, the new methods Mosaic and Self-Adversarial Training (SAT) have been applied. Further developments in the field focus on the semantic distribution in the dataset, thus facilitating label smoothing to increase model robustness. Ultimately, the model implements DropBlock as a new state-of-art regularization method. In the "bag of specials", highly specialized techniques are introduced which give the algorithm its final edge in accuracy. For example, attention mechanisms like Spatial Attention Module (SAM) were introduced for point-wise attention, as it has been shown to improve accuracy at minimal cost of computing power. Eventually, only one single GPU is needed to train the YOLO v4 model.

4.3 YOLO v5 (no paper published yet)

With some major improvements, this one poses one of the most famous algorithms for real-time image detection. In a supplementary Python notebook, a possible implementation of the YOLO v5 model is illustrated, with the detection of people wearing and not wearing masks. The dataset was taken from Kaggle and uploaded into Roboflow. Note how the labels of the training images were unbalanced, but as a first estimation of the model, data augmentation was not performed.

As YOLO v5 is a single-stage object detector, it has three important parts like any other single-stage object detector.

1) Model Backbone
2) Model Neck
3) Model Head

Model Backbone is mainly used to extract important features from the given input image. In YOLO v5 the CSP — Cross Stage Partial Networks are used as a backbone to extract rich in informative features from an input image.

Model Neck is mainly used to generate feature pyramids. Feature pyramids help models to generalize well on object scaling. It helps to identify the same object with different sizes and scales.

Feature pyramids are very useful and help models to perform well on unseen data. In YOLO v5 PANet is used for as neck to get feature pyramids.

The model Head is mainly used to perform the final detection part. It applies anchor boxes on features and generates final output vectors with class probabilities, objectness scores, and bounding boxes.

In YOLO v5, model head is the same as the previous YOLO V3 and V4 versions.

In YOLO v5, the Leaky ReLU activation function is used in middle/hidden layers and the sigmoid activation function is used in the final detection layer.

Reference List

Bharath Raj (2018). *A Simple Guide to the Versions of the Inception Network*. [online] Medium. Available at: https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202 [Accessed 13 Dec. 2020].

Github User mihir135 (2020). *YOLOv5-s Model Architecture*. [online] Gist. Available at: https://gist.github.com/mihir135/2e5113265515450c8da934e15d97fc6b [Accessed 13 Dec. 2020].

Larxel (2020). *Face Mask Detection*. [online] Kaggle.com. Available at: https://www.kaggle.com/andrewmvd/face-mask-detection [Accessed 13 Dec. 2020].

Math-dude (2019). *How do custom input_shape for Inception V3 in Keras work?* [online] Stack Overflow. Available at: https://stackoverflow.com/questions/54522336/how-do-custom-input-shape-for-inception-v3-in-keras-work [Accessed 13 Dec. 2020].

Sik-Ho Tsang (2018). *Review: Inception-v3 — 1st Runner Up (Image Classification) in ILSVRC 2015*. [online] Medium. Available at: https://sh-tsang.medium.com/review-inception-v3-1st-runner-up-image-classification-in-ilsvrc-2015-17915421f77c [Accessed 13 Dec. 2020].

Srivastava, N., Hinton, G., Krizhevsky, A. and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, [online] 15, pp.1929–1958. Available at: https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

Towards AI Team (2020). *YOLO V5—Explained and..*. [online] Towards AI — The Best of Tech, Science, and Engineering. Available at: https://towardsai.net/p/computer-vision/yolo-v5%E2%80%8A-%E2%80%8Aexplained-and-demystified [Accessed 13 Dec. 2020].