Report Homework 2 – Machine Learning

Denise Landini

1938388

Sapienza
Università di Roma

Summary

Chapter 1: Project Overview

In this homework the objective is to provide a solution for the classification of images regarding objects in a home environment.

1.1 Dataset

The dataset supplied to me presents eight classes:

- Energy_drink;
- Forks_Lemons;
- Lollipops;
- Wet_Mops;
- dinner_plat;
- pickled_vegetables;
- plastic_tray.

The dataset is composed by 9168 images.

1.2 Import libraries and define functions

For libraries I decide to use **Keras** and **Tensorflow**, so I import them.

I have also decided to define some functions which are useful to further evaluate and for train all models that I used.

In particular, I have:

- *plot_confusion_matrix*: allows me to print and plot the confusion matrix;
- *savemodel*: allows me to save the model used after trained it for 10 epochs. It is very useful because I can understand better the evolution of the accuracy and of the other important parameters. Furthermore, if I save the model, I can reload the model whenever I want and train it again;
- *savehistory*: allows me to save the history of the model;
- *loadmodel*: allows me to load the previously trained model for train it for another 10 epochs;
- *loadhistory:* allows me to load the previously history of the model.

Chapter 2: Load and Split Data

First of all, it is important to import *ImageDataGenerator* to take the images from the dataset and pre-process them. This library of Keras allow us to use data augmentation automatically when we train a model. The class are instantiated and set up for the configuration of data augmentation specified by the arguments to the class constructor.

I decide to use the parameter *validation_split*, to subdivide the dataset into 80% train and 20% test. So, I have 7338 training samples and 1830 test samples.

Then, I use the *flow_from_directory* method to take the dataset and I apply this function to create:

- **train_generator** in which I set the parameter *subset = 'training'* because it identifies that is just the train generator and I set the parameter *shuffles = 'True'* because I want to shuffle the order of the image that is being yielded.
- **validation_generator** in which I set the parameter *subset = 'validation'* and I set the parameter *shuffles = 'False'*, so the dataset is sorted in alphanumeric order.

In both cases I want to specify the target_size, that is the dimensions to which all images will be resized, and I choose *target_size = (256,256)*, but I decide to not set the batch_size because I think that the default *batch_size = 32* is fine.
Furthermore, I set *class_mode = 'categorical'* because I have more than two classes to predict.

Chapter 3: Models used

In deep learning, a CNN (Convolutional Neural Network) is a class of deep neural networks, that is used to analyse the images.

A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume through function. In my case the input is given by *input_shape* that is (256,256,3) because I have the size of the images considered plus the factor 3 because I set *color_mode = 'rgb'*.

I choose to compare four different models:

- LeNet model;
- AlexNet model;
- TransferNet model;
- DeniNet model.

For each type of model, I consider different parameters and I will explain the architecture of these models in the following subchapter.

3.1 LeNet model

The architecture is composed by **seven layers** and in this case, I consider LeNet using the hyperbolic tangent as activation function. The input images are 256 x 256 x 3.

- In the **first** convolutional layer I have 6 kernels of size 5 x 5;
- The **second** layer is the pooling layer, in which I have size 2 x 2 and stride 2 x 2;
- In the **third** layer I have 16 kernels of size 5 x 5 and strides 1 x 1;
- The **fourth** layer is the pooling layer with the same parameters of the second layer;
- In the **fifth** convolutional layer I have 120 kernels of size 5 x 5 and strides 1 x 1;
- Then I flat the model in order to pass it to a fully connected layer.

At this point I have **2 fully connected layers**:

- the first with 84 units and the second with 10 units;
- In the last layer I use the non-linear activation function *softmax*.

I also decide to insert LeNet model with **ReLU** as activation function instead of **hyperbolic tangent** and in the chapter 6 I describe the benefit that I have had.
The optimizer that I use is in both case the **adam optimizer**.

3.2 AlexNet model

The architecture is composed by **eight layers**, in which **five are convolutional layers** and **three are full-connected layers** plus **output layer.**

In the convolutional layers AlexNet use the **ReLU** (Rectified Linear Unit) activation function which flattens the response to all negative values to zero, while leaving everything unchanged for values equal or grater than zero.

- The **first** convolutional layer filters the 256 x 256 x 3 input image with 96 kernels of size 11 x 11 x 3 with a stride of 2 x 4 pixels, where the stride is the distance between the receptive files centres of neighbouring neurons in a kernel map and padding 0. The output is 54 x 54 x 96, but to this we overlap a *MaxPooling* layer having dimensions 2 x 2 with stride 2 x 2 to reduce the size to 27 x 27 x 96 and finally we apply the *BatchNormalization* before passing it to the next layer;
- The **second** convolutional layer filters the output of the first convolutional layer with 256 kernels of size 11 x 11 x 3 with strides of 1 x 1. We use the same parameters that we use in the first convolution and I obtain 8 x 8 x 256 for the output;

- The **third** convolutional layer filters the output of the second convolutional layer with 384 kernels of size 3 x 3 x 3 with strides of 1 x 1. In this case, I don't apply the *MaxPooling,* but I do only the *BatchNormalization*. The output is 6 x 6 x 384;
- The **fourth** convolutional layer filters the output of the third convolutional layer with the same parameters that I have used for the third convolution. The output is 4 x 4 x 384;
- The **fifth** convolutional layer filters the output of the fourth convolutional layer with the same parameters of the third convolution, but I add the *MaxPooling* of size 2 x 2 and strides 2 x 2. The output is 1 x 1 x 256, after the *BatchNormalization*;
- Then I flat the model in order to pass it to a fully connected layer;
- At this point I have **3 dense layers** for each layer I use ReLU as activation function and I add *dropout* to prevent the overfitting. Finally, for the **output layer** there is a *softmax* output layer.

3.3 Pre-trained model: VGG-16

I decide to introduce a **pre-trained model** in my homework because they represent some advantages respect to the not pre-trained model. In fact, a pre-trained model is a model that has been previously trained on a dataset and contains the weights and biases that represent the characteristics of whichever dataset it was trained on. The most important advantage of pre-trained model is that if you use these models you can save your time.

In particular, I choose the **VGG-16** pre-trained model.

In my opinion is interesting to compare this pre-trained model with AlexNet because it makes some improvements respect to the AlexNet. In fact, it replaces the large kernel size filters of AlexNet, that are 11 for example in the first convolutional layer, with multiple 3 x 3 kernel-sized filters one after another.

In this model, I have a lot of layers that are follow with full-connected layers. As activation function, it uses **ReLU** function, except for the last layer in which use *softmax*. Finally, it puts *two dropouts* to prevent the model to the problem of overfitting, or it try to decrease its presence.
It has also *BatchNormalization* layers to improve performance model.

3.4 DeniNet model

I want to create my net with **six convolutional layers**, **two dense layers** and the **output layer**.

In particular, I want to increase the number of layers with respect to AlexNet and I also change the size of the kernel filters and I select the ReLU as activation function for the reasons that I will explain in the 6.1 paragraph.

- In the **first** convolutional layer I filter the input shape with 15 kernel filters with size 4 x 4.

After the first convolutional layer, for each of the remaining convolutional layers, I introduce also the *MaxPooling* that reduce the size of the matrix and in this case, I consider the size 2 x 2.

- In the **second** convolutional layer I decide to increase the number of filters and I filter the output of the first convolutional layer with 20 kernel filters with size 4 x 4;
- In the **third** convolutional layers I decide to increase both the number of filters and the size of the kernel;
  So, I decide to insert 30 kernels of size 4 x 4;
- I do the same for the **fourth** and **sixth** convolutional layers;
- In the **fifth** convolutional layer I decide to unchanged the number of kernels, but I try to reduce the size to 3 x 3;
- Then I flat the model in order to pass it to a fully connected layer;
- I have the fully connected NN in which I add other **2 dense layers** followed by *dropout*, which is set in order to avoid the overfitting problem;
- The important thing is that the **output layer** must have exactly the number of classes of my dataset (eight) as the unit number and for this reason the activation function must be *softmax*, also because I have a multiclass problem.

As optimizer I want to change from the one used by LeNet and AlexNet, so I decide to use *rmsprop*.

The model ends when the compile function is executed.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 253, 253, 15)      735
_________________________________________________________________
activation (Activation)      (None, 253, 253, 15)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 250, 250, 20)      4820
_________________________________________________________________
activation_1 (Activation)    (None, 250, 250, 20)      0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 125, 125, 20)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 121, 121, 30)      15030
_________________________________________________________________
activation_2 (Activation)    (None, 121, 121, 30)      0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 60, 60, 30)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 57, 57, 30)        14430
_________________________________________________________________
activation_3 (Activation)    (None, 57, 57, 30)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 30)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 26, 26, 30)        8130
_________________________________________________________________
activation_4 (Activation)    (None, 26, 26, 30)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 13, 13, 30)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 10, 10, 30)        14430
_________________________________________________________________
activation_5 (Activation)    (None, 10, 10, 30)        0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 5, 5, 30)          0
_________________________________________________________________
flatten (Flatten)            (None, 750)               0
_________________________________________________________________
dense (Dense)                (None, 128)               96128
_________________________________________________________________
activation_6 (Activation)    (None, 128)               0
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 96)                12384
_________________________________________________________________
activation_7 (Activation)    (None, 96)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 96)                0
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 776
_________________________________________________________________
activation_8 (Activation)    (None, 8)                 0
=================================================================
```

1) DeniNet

Chapter 4: Train the model

The function **fit_generator** is used to train the model that I have chosen.

This fit function takes as input:

- *Train_generator*;
- *Verbose*: that I set to one in order to have an animated progress bar;
- *Epochs*: is a number of epochs, so the number of iterations that I decide to set to 10. As I mentioned before, I choose this low value because I want to control the evolution of the important evaluation parameters (i.e., accuracy and loss) in detail;
- *Steps_per_epochs*: is the total number of batch iterations before a training epoch is considered finished and starting the next epoch. I choose to initialize it as train_generator.n//train_generator.batch_size (n = number of samples);
- *Validation data*: is the train_generator;
- *Validation_steps*: is the total number of steps to yield from validation_data generator before stopping at the end of every epoch.

As I mentioned before, after the training of the model, I save the model and the history.

Chapter 5: Evaluation of the model and metrics

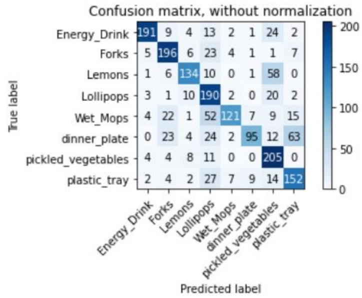
For models I chose to show the data that I have obtained with:

- **Classification report**: that allows me to see very precisely the accuracy, but also the Precision, Recall and F1-score. More specifically, with this representation I can see these parameters for each of the eight classes;
- **Confusion matrix**: that allows me to visualize immediately the classes that are more confused;
- **Plot of accuracy and loss**: is a graphic in which we have the function of accuracy and the function of loss.
    - *Accuracy* can show us if a model is being trained correctly and how it may perform generally. However, in some cases, accuracy only is not enough to assess the performance of a classification method. In fact, it does not do well when we have an unbalanced data set;
    - *Loss function* is used to optimize the parameter values in a NN model. It maps a set of parameter values for the network on a scalar value that indicates how well these parameters do the work that the network is intended to do.

5.1AlexNet: results

	precision	recall	f1-score	support
Energy_Drink	0.930	0.756	0.834	246
Forks	0.758	0.774	0.766	243
Lemons	0.816	0.633	0.713	210
Lollipops	0.550	0.846	0.667	228
Wet_Mops	0.893	0.541	0.674	231
dinner_plate	0.865	0.430	0.575	223
pickled_vegetables	0.597	0.905	0.719	232
plastic_tray	0.604	0.737	0.664	217
accuracy			0.705	1830
macro avg	0.751	0.703	0.701	1830
weighted avg	0.753	0.705	0.704	1830

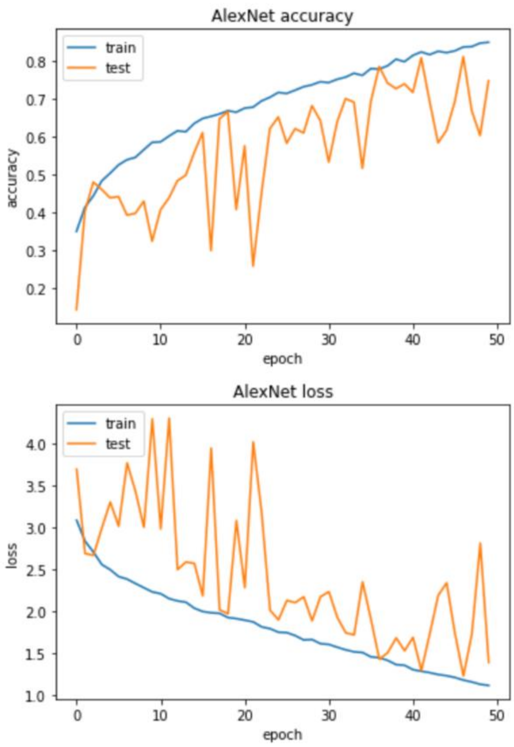
2) Classification report AlexNet



3) Confusion matrix AlexNet

True		Predicted	errors	err %
dinner_plate	->	plastic_tray	63	0.69 %
Lemons	->	pickled_vegetables	58	0.63 %
Wet_Mops	->	Lollipops	52	0.57 %
plastic_tray	->	Lollipops	27	0.29 %
Energy_Drink	->	pickled_vegetables	24	0.26 %
dinner_plate	->	Lollipops	24	0.26 %
dinner_plate	->	Forks	23	0.25 %
Forks	->	Lollipops	23	0.25 %
Wet_Mops	->	Forks	22	0.24 %
Lollipops	->	pickled_vegetables	20	0.22 %
Wet_Mops	->	plastic_tray	15	0.16 %
plastic_tray	->	pickled_vegetables	14	0.15 %
Energy_Drink	->	Lollipops	13	0.14 %
dinner_plate	->	pickled_vegetables	12	0.13 %
pickled_vegetables	->	Lollipops	11	0.12 %
Lollipops	->	Lemons	10	0.11 %
Lemons	->	Lollipops	10	0.11 %
Wet_Mops	->	pickled_vegetables	9	0.10 %
plastic_tray	->	dinner_plate	9	0.10 %
Energy_Drink	->	Forks	9	0.10 %
pickled_vegetables	->	Lemons	8	0.09 %
plastic_tray	->	Wet_Mops	7	0.08 %
Forks	->	plastic_tray	7	0.08 %
Wet_Mops	->	dinner_plate	7	0.08 %
Lemons	->	Forks	6	0.07 %
Forks	->	Lemons	6	0.07 %
Forks	->	Energy_Drink	5	0.05 %
dinner_plate	->	Lemons	4	0.04 %
Forks	->	Wet_Mops	4	0.04 %
pickled_vegetables	->	Energy_Drink	4	0.04 %
plastic_tray	->	Forks	4	0.04 %
Wet_Mops	->	Energy_Drink	4	0.04 %
Energy_Drink	->	Lemons	4	0.04 %
pickled_vegetables	->	Forks	4	0.04 %
Lollipops	->	Energy_Drink	3	0.03 %
Energy_Drink	->	Wet_Mops	2	0.02 %
plastic_tray	->	Lemons	2	0.02 %
plastic_tray	->	Energy_Drink	2	0.02 %
Lollipops	->	plastic_tray	2	0.02 %
Energy_Drink	->	plastic_tray	2	0.02 %
dinner_plate	->	Wet_Mops	2	0.02 %
Lollipops	->	Wet_Mops	2	0.02 %
Forks	->	dinner_plate	1	0.01 %
Forks	->	pickled_vegetables	1	0.01 %
Wet_Mops	->	Lemons	1	0.01 %
Lemons	->	Energy_Drink	1	0.01 %
Lemons	->	dinner_plate	1	0.01 %
Energy_Drink	->	dinner_plate	1	0.01 %
Lollipops	->	Forks	1	0.01 %

4) Percentage error AlexNet



5) Plot AlexNet

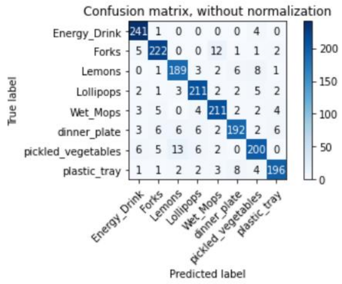
I train this model 50 times, and I do not obtain a perfect accuracy, but it is good, as you can see in the classification report and in the confusion matrix. In the plot, I can see that for the accuracy, the train function is increasing more and more according to the decreasing of the loss. But I have a different behaviour for the test set, that present a lot of peaks in accuracy and loss.

It works well with Energy_drink, but not very well with the other classes and it often confuse the dinner_plate with plastic_tray. It is reasonable because they are both containers and If they have the same colour they can be easily confused.

5.2 TrasnferNet: results

	precision	recall	f1-score	support
Energy_Drink	0.923	0.980	0.951	246
Forks	0.917	0.914	0.915	243
Lemons	0.887	0.900	0.894	210
Lollipops	0.909	0.925	0.917	228
Wet_Mops	0.902	0.913	0.908	231
dinner_plate	0.910	0.861	0.885	223
pickled_vegetables	0.885	0.862	0.873	232
plastic_tray	0.929	0.903	0.916	217
accuracy			0.908	1830
macro avg	0.908	0.907	0.907	1830
weighted avg	0.908	0.908	0.908	1830

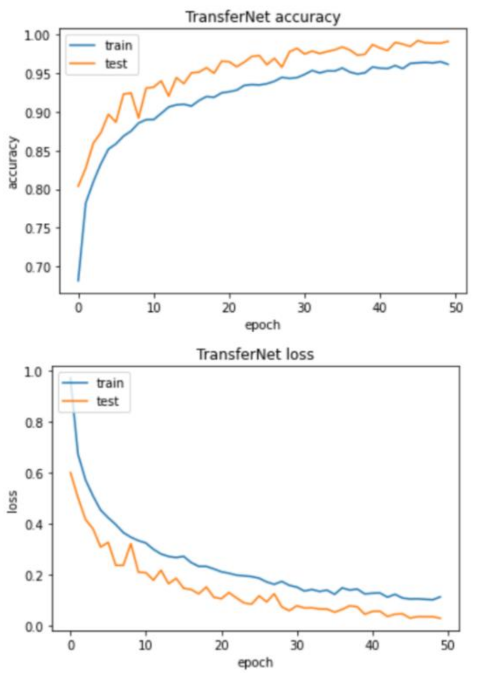
6) *Classification report TransferNet*



7) *Confusion matrix TransferNet*

True		Predicted	errors	err %
pickled_vegetables	->	Lemons	13	0.14 %
Forks	->	Wet_Mops	12	0.13 %
plastic_tray	->	dinner_plate	8	0.09 %
Lemons	->	pickled_vegetables	8	0.09 %
dinner_plate	->	Lollipops	6	0.07 %
pickled_vegetables	->	Lollipops	6	0.07 %
dinner_plate	->	Forks	6	0.07 %
pickled_vegetables	->	Energy_Drink	6	0.07 %
dinner_plate	->	plastic_tray	6	0.07 %
dinner_plate	->	Lemons	6	0.07 %
Lemons	->	dinner_plate	6	0.07 %
Lollipops	->	pickled_vegetables	5	0.05 %
pickled_vegetables	->	Forks	5	0.05 %
Forks	->	Energy_Drink	5	0.05 %
Wet_Mops	->	Forks	5	0.05 %
Wet_Mops	->	plastic_tray	4	0.04 %
Wet_Mops	->	Lollipops	4	0.04 %
plastic_tray	->	pickled_vegetables	4	0.04 %
Energy_Drink	->	pickled_vegetables	4	0.04 %
plastic_tray	->	Wet_Mops	3	0.03 %
Wet_Mops	->	Energy_Drink	3	0.03 %
Lollipops	->	Lemons	3	0.03 %
dinner_plate	->	Energy_Drink	3	0.03 %
Lemons	->	Lollipops	3	0.03 %
Lollipops	->	Wet_Mops	2	0.02 %
plastic_tray	->	Lollipops	2	0.02 %
plastic_tray	->	Lemons	2	0.02 %
pickled_vegetables	->	Wet_Mops	2	0.02 %
Forks	->	plastic_tray	2	0.02 %
Lollipops	->	dinner_plate	2	0.02 %
dinner_plate	->	pickled_vegetables	2	0.02 %
Lemons	->	Wet_Mops	2	0.02 %
Lollipops	->	Energy_Drink	2	0.02 %
Wet_Mops	->	pickled_vegetables	2	0.02 %
Wet_Mops	->	dinner_plate	2	0.02 %
Lollipops	->	plastic_tray	2	0.02 %
dinner_plate	->	Wet_Mops	2	0.02 %
Lemons	->	Forks	1	0.01 %
Lemons	->	plastic_tray	1	0.01 %
Forks	->	pickled_vegetables	1	0.01 %
Forks	->	dinner_plate	1	0.01 %
plastic_tray	->	Energy_Drink	1	0.01 %
plastic_tray	->	Forks	1	0.01 %
Lollipops	->	Forks	1	0.01 %
Energy_Drink	->	Forks	1	0.01 %

8) *Percentage error TransferNet*



9) *Plot TransferNet*

I train this model 50 times, and I obtain a very good accuracy because is close to 1. This means that TransferNet can correctly classify the images that are in the dataset, especially for Energy_drink. I can observe this behaviour both in classification report and confusion matrix. But, also in the plot I can see that both train and test function as the same trend. How I can see from the table that represent the percentage of error, the column of errors has a very low value.

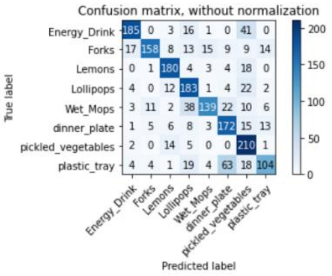
In fact, they start with an average accuracy (0.65/0.80) at the first training and then they increase it in a logarithmic way.
For loss we have the opposite behaviour, so they start with an average loss (1.00/0.60) at the first training and then they decrease it, and after 50 epochs of training they has respectively 0.18 and 0.08 of loss.

Comparison: respect to the AlexNet this model is better because the accuracy is better, but also because in the graphic plot there is not all the peaks that there are in the AlexNet.

5.3 DeniNet: results

	precision	recall	f1-score	support
Energy_Drink	0.856	0.752	0.801	246
Forks	0.883	0.650	0.749	243
Lemons	0.796	0.857	0.826	210
Lollipops	0.640	0.803	0.712	228
Wet_Mops	0.837	0.602	0.700	231
dinner_plate	0.628	0.771	0.692	223
pickled_vegetables	0.612	0.905	0.730	232
plastic_tray	0.743	0.479	0.583	217
accuracy			0.727	1830
macro avg	0.749	0.727	0.724	1830
weighted avg	0.751	0.727	0.725	1830

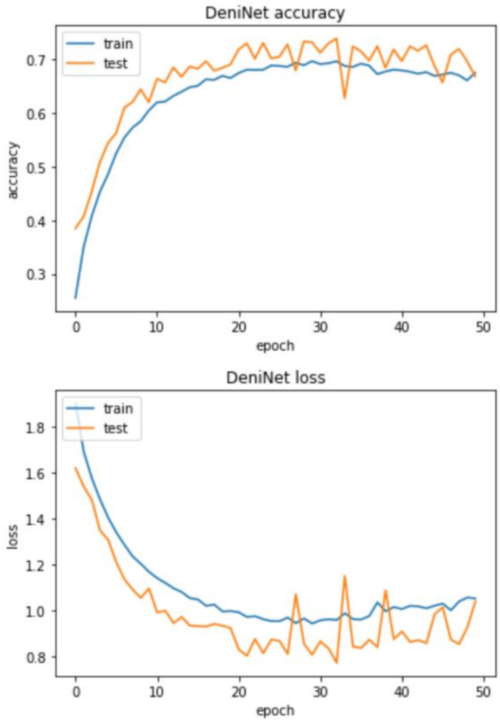
10) Classification report DeniNet



11) Confusion matrix DeniNet

True		Predicted	errors	err %
plastic_tray	->	dinner_plate	302	3.29 %
Energy_Drink	->	pickled_vegetables	179	1.95 %
Lollipops	->	pickled_vegetables	156	1.70 %
Lemons	->	pickled_vegetables	112	1.22 %
Wet_Mops	->	pickled_vegetables	110	1.20 %
plastic_tray	->	pickled_vegetables	108	1.18 %
Wet_Mops	->	Lollipops	107	1.17 %
Forks	->	Energy_Drink	96	1.05 %
Forks	->	Lemons	88	0.96 %
Forks	->	pickled_vegetables	86	0.94 %
Forks	->	dinner_plate	79	0.86 %
Forks	->	Lollipops	74	0.81 %
dinner_plate	->	pickled_vegetables	74	0.81 %
pickled_vegetables	->	Lemons	72	0.79 %
Wet_Mops	->	dinner_plate	69	0.75 %
Lollipops	->	Lemons	67	0.73 %
Wet_Mops	->	Forks	63	0.69 %
plastic_tray	->	Lollipops	62	0.68 %
Forks	->	Wet_Mops	61	0.67 %
Wet_Mops	->	Energy_Drink	56	0.61 %
Forks	->	plastic_tray	53	0.58 %
dinner_plate	->	plastic_tray	52	0.57 %
Wet_Mops	->	plastic_tray	51	0.56 %
Energy_Drink	->	Lollipops	50	0.55 %
dinner_plate	->	Lemons	43	0.47 %
Lollipops	->	Energy_Drink	38	0.41 %
plastic_tray	->	Forks	36	0.39 %
Wet_Mops	->	Lemons	35	0.38 %
dinner_plate	->	Lollipops	34	0.37 %
plastic_tray	->	Energy_Drink	29	0.32 %
Lemons	->	Lollipops	27	0.29 %
Lollipops	->	Forks	26	0.28 %
pickled_vegetables	->	Lollipops	24	0.26 %
Lollipops	->	dinner_plate	23	0.25 %
plastic_tray	->	Lemons	22	0.24 %
dinner_plate	->	Forks	15	0.16 %
pickled_vegetables	->	Energy_Drink	14	0.15 %
Lollipops	->	Wet_Mops	13	0.14 %
dinner_plate	->	Wet_Mops	12	0.13 %
Energy_Drink	->	Forks	12	0.13 %
Lollipops	->	plastic_tray	12	0.13 %
Lemons	->	dinner_plate	11	0.12 %
plastic_tray	->	Wet_Mops	11	0.12 %
dinner_plate	->	Energy_Drink	10	0.11 %
Energy_Drink	->	Lemons	9	0.10 %
Energy_Drink	->	plastic_tray	8	0.09 %
Lemons	->	Wet_Mops	7	0.08 %
pickled_vegetables	->	dinner_plate	6	0.07 %
Lemons	->	Energy_Drink	4	0.04 %
pickled_vegetables	->	plastic_tray	4	0.04 %
Lemons	->	Forks	3	0.03 %
Energy_Drink	->	Wet_Mops	3	0.03 %
pickled_vegetables	->	Forks	2	0.02 %
Energy_Drink	->	dinner_plate	2	0.02 %
Lemons	->	plastic_tray	1	0.01 %

12) Percentage error DeniNet



13) Plot DeniNet

I train this model 50 times, and I obtain an enough good accuracy. The trend is similar to the previous one, but the most important change is that the test function both in accuracy and loss has some peaks.

I can also see that in the last part of the graphic the loss of the test set and train set is increasing, and the accuracy of the train set is constant, so I think that there is low overfitting.
How I can see from the percentage error, this network misclassified plastic_tray with dinner_plate for the reason that I explain in AlexNet. But it also misclassified a lot of classes as pickled_vegetables.

Comparisons: Respect to AlexNet I have less peaks in the test function, but respect to the TransferNet I have more peaks, so I can conclude that this network is between these two types of CNN. I think that this behaviour is due to the increase of the convolutional layers respect to AlexNet and for the variation of the number of kernel filters and their size.

Chapter 6: Comparisons with activation functions

6.1 Comparison between LeNet with tanh and LeNet with ReLU

I think that is interesting to compare LeNet with two different activation functions:

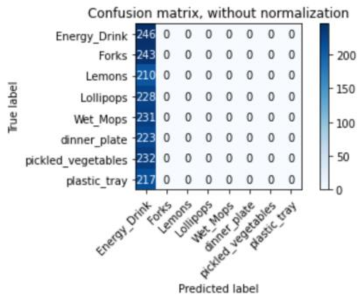
- *tanh*;
- *ReLU*.

Tanh is a non-linear activation function that has the problem that:

- it becomes saturated (in fact the derivative is not monotony), so this means that the large values snap to 1 or 0. Once saturated, it becomes difficult for the learning algorithm to improve the performance of the model;
- it is sensitive to changes around their mid-point of their input, such as 0.0;
- it requires an exponential calculus.

	precision	recall	f1-score	support
Energy_Drink	0.134	1.000	0.237	246
Forks	0.000	0.000	0.000	243
Lemons	0.000	0.000	0.000	210
Lollipops	0.000	0.000	0.000	228
Wet_Mops	0.000	0.000	0.000	231
dinner_plate	0.000	0.000	0.000	223
pickled_vegetables	0.000	0.000	0.000	232
plastic_tray	0.000	0.000	0.000	217
accuracy			0.134	1830
macro avg	0.017	0.125	0.030	1830
weighted avg	0.018	0.134	0.032	1830

14) Classification Report LeNet with tanh



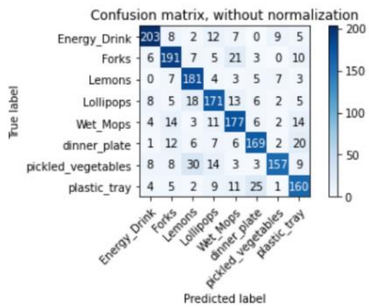
15) Confusion matrix LeNet with tanh

ReLU is the Rectified Linear Unit and has more advantages than tanh:

- it is simply to compute because it requires only a max() function;
- it has value equal to zero for the negative x and has value x for positive values of x;
- it avoids the problem of vanishing gradients, as the gradients remain proportional to the node activations. In fact, calculating the derivative is very simple: for all negative values it is equal to zero, while for the positive ones it is equal to 1. In the origin the derivative is indefinite, but it is still set to zero by convention.

	precision	recall	f1-score	support
Energy_Drink	0.868	0.825	0.846	246
Forks	0.764	0.786	0.775	243
Lemons	0.727	0.862	0.789	210
Lollipops	0.734	0.750	0.742	228
Wet_Mops	0.734	0.766	0.750	231
dinner_plate	0.779	0.758	0.768	223
pickled_vegetables	0.872	0.677	0.762	232
plastic_tray	0.708	0.737	0.722	217
accuracy			0.770	1830
macro avg	0.773	0.770	0.769	1830
weighted avg	0.775	0.770	0.770	1830

16) Classification Report LeNet with ReLU



17) Confusion matrix LeNet with ReLU

So, the best performance is given by the LeNet with ReLU for the reasons that I explain above. In fact, I can see in red rectangle the big difference in F1-Score between using tanh or ReLU.

As you can see in the figure, LeNet with ReLU has an high accuracy (0.770), instead of LeNet with tanh, in which the accuracy is very high (0.134)

Chapter 7: Conclusion

CNN	Accuracy
LeNet with tanh	**0.134**
LeNet with ReLU	0.770
AlexNet	0.705
TransferNet	**0.908**
DeniNet	0.727

I think that TrasferNet has the highest accuracy because is a pre-trained model and it is used for images recognition, but I think that I have obtained a good result also by applying LeNet with ReLU and DeniNet.

The worst accuracy that I have obtained is with LeNet with tanh as activation function, but I expected this result because of the property that has the hyperbolic tangent function. In particular, all the classes are classified as Energy_drink, but this behaviour is not acceptable (i.e., it is strange to confuse a dinner_plate with an Energy_drink).

For AlexNet, DeniNet and LeNet with ReLU the higher percentage of error always occurs with the wrong classification of dinner_plate with plastic_tray and vice-versa, that is reasonable. For the other classes they have a good classification, but not perfect.

I think that if I had trained more LeNet with ReLU and TrasferNet, the accuracy would have continued to grow because the trend of the functions is regular, but I don't do that because it also can have overfitting. While I think that with DeniNet the achieved accuracy is the highest I could get (for the plot seen above).